A Framework for Generic and Energy Efficient Context Recognition for Personal Mobile Devices

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften
(Dr. rer. nat.)

durch die Fakultät für Wirtschaftswissenschaften der
Universität Duisburg-Essen
Campus Essen

vorgelegt von
Muhammad Umer Iqbal

geboren in Karachi

Essen, 2015

Tag der mündlichen Prüfung: 12.4.2016

Erstgutachter:	Prof. Dr. rer. nat. Pedro José Marrón
Zweitgutachter:	Prof. Dr. rer. nat. Gregor Schiele

Contents

List of Figures

List of Tables

1 ABSTRACT

The advancements in the field of mobile computing over the last decade have enabled the scientific community to expedite the theoretical and experimental work to achieve the vision of ubiquitous computing. As ubiquitous computing aims to provide seamless and distraction free task support to its users, one of the essential pieces of information required by the ubiquitous computing systems to do so is the context of its users. Context of a user can be defined as the information that describes the task the user is performing and the environment in which the user is currently present. Among various platforms that are commonly used to determine user's context, the personal mobile devices like smart phones stand out as one of the most widely used and widely evaluated ones.

However, despite numerous advantages that are provided by modern day personal mobile devices, such as high computational and communication capabilities, variety of onboard sensors to capture raw data related to user's motion and environment, high resolution displays to enable interaction with other services and systems, these devices suffer from limited battery resources. In contrast to the advancements in other domains, the advancements in the battery domain have not been up to the mark. Consequently, the context recognition applications developed for these devices suffer from the trade-off between achieving accuracy and longevity of other device's basic operations. As a result, most of the existing context recognition applications for these devices are fine tuned for specific context types and thereby lacks generality. The situation gets worse when a number of context recognition applications are executed simultaneously, thus competing for limited resources and consuming the device's battery additively.

To address the aforementioned issues, this thesis provides a generic and energy efficient context recognition framework for personal mobile devices. The main contribution consists of a generic framework to support development of context recognition applications supported by algorithms to achieve their energy efficient execution. The proposed framework consists of two systems namely the component system and the activation system. The component system allows developers to create context recognition applications using a component abstraction. This enables runtime analysis of applications' structures to adopt our novel energy efficiency mechanism. The activation system uses a state machine abstraction to allow context dependent activation of context recognition configurations pertaining to relevant user's tasks such that only needed configurations are executed to determine only the relevant context characteristics, thereby enabling energy efficiency. The activation system also provides generic applicability of four different energy efficiency techniques, already used in different existing systems but mostly for specific context characteristics.

To aid rapid prototyping, both systems are equipped with off-line development tools. The tools include graphical editors and a component tool-kit. The graphical editors allow developers to create component configurations used by the component system and state machines used by the activation system. These editors enable developers to create component configurations and state machines by simply dragging, dropping and connecting different models used in our component and state machine abstractions. These tools also provide validation and code generation utilities. In addition to the graphical editor, the framework provides a component tool-kit which consists of a number of already implemented sensing, preprocessing and classification components which can be re-used in new applications.

In order to provide the energy efficient execution of context recognition applications, the thesis introduces a novel energy efficiency technique called configuration folding. Configuration folding analyses structures of simultaneously executing context recognition applications to identify redundant functionalities between them and as an output produces a single redundancy free context recognition configuration which holds the structural integrity of all applications. Consequently, the overall energy expenditure is reduced compared to the original expenditure when redundant functionalities are not removed. The experimental evaluation of configuration folding on test applications shows energy savings between 13 and 48 %. The thesis also investigates optimization possibilities in configuration folding in case the redundant functionalities between the applications differ in parametrization. Towards this end, the thesis identifies commonly used parameters in context recognition applications and defines relations between them. Finally, an extended version of configuration folding is introduced to handle the differences in parametrization. The evaluation of the extended version of configuration folding on test scenarios shows energy saving of up to 45%.

The contributions in this thesis have been evaluated extensively. The framework has been used in number of European Commission (EC) projects and in student projects and theses at the University of Duisburg-Essen, Germany. Using the component system and the activation system, a number of applications have been developed in those projects. Some of these applications include crowd density estimation in buses, bus ride detections, navigation application for buses in Madrid, user movement detection, user localization, fall detection application etc. Moreover, the component system, the activation system and the configuration folding technique have been published in different prestigious conferences and workshops.

2 INTRODUCTION

Ubiquitous computing systems envision seamless and distraction free support for the everyday tasks of their users. In order to fulfil this vision, one of the important pieces of information required by the ubiquitous computing systems is the context of the user. The context of the user can be defined as the information that describes the task the user is performing and the environment in which the user is currently present. Among various platforms for determining user context, personal mobile devices such as smart phones are widely used these days. However, due to the limited battery resources in these devices, their potential for determining context continuously with high accuracy is compromised. In this thesis, we present a generic and energy efficient context recognition framework for personal mobile devices, particularly for smart phones. The framework allows development and execution of context recognition applications together with off-line tools to aid rapid prototyping. The framework is equipped with energy efficiency techniques with which the energy expenditure of executing context recognition applications on smart phones is reduced. This chapter provides the motivation and contribution of the thesis followed by the overview of the remainder of the thesis.

2.1 Motivation

Ubiquitous computing as envisioned by Mark Weiser [Wei99] enable users to carry out their everyday tasks with seamless and distraction free support from the technologies amalgamated in the user's environment. The vision enables different electronic devices to autonomously interact with each other and perceive the environment of the user and translate it into meaningful information called context. Examples of ubiquitous computing systems could be a system which automatically adjusts the intensity of lights in the offices when a user is working or a system which guide users to park their cars in the parking places. For either of these or many other examples, ubiquitous computing systems rely on the context information of the user. Consequently, ubiquitous computing systems require suitable (in terms of cost and usability) technological means to gather the context information.

Over the last decade, there has been tremendous advancements in the field of micro electronics which has resulted in the wide spread adoption of computationally powerful yet miniature devices by the general public in their daily lives. Consequently, the research and development in ubiquitous computing paradigms have received increased attention from the scientific community. As a result, new concepts have been presented, new systems

and applications have been developed and ubiquitous computing related systems and tools have become a major business throughout the world.

Among various platforms for context recognition that exist today, personal mobile devices such as smart phones, tablets etc. are among the most widely used and most thoroughly evaluated ones. The ubiquity of personal mobile devices provides a promising technical basis to determine the context of users in an automated manner on a large scale. The reason for this is threefold. First and foremost, a personal mobile device is directly associated with a particular user. Since the user carries the device continuously during the day, many features of the device context can be used as a representation of the user context without further ado. Secondly, more and more devices are equipped with sensors. Examples of such sensors are gyroscopes, accelerometers, cameras and microphones, to name a few. At the same time, an increasing number of devices are able to use wireless communication technology such as General Packet Radio Service (GPRS) or Universal Mobile Telecommunications System (UMTS) to access relevant on-line information sources like calendars, task lists or maps at any point in time. Together, this creates an unprecedented richness of physical and virtual information sources that can be tapped to determine the context of a user seamlessly. Last but not least, although personal mobile devices are often considered resource-poor when compared to traditional computers, the available resources are usually underutilized in normal operation.

Due to these factors, researchers and practitioners have developed a number of context recognition systems and applications for personal mobile devices for different domains such as physical activity recognition [BI04], [LCB06], traffic monitoring [MPR08], social networking [MLEC07], healthcare [Bar04a], [MGP+12] etc. Such applications are usually fine-tuned to recognize the set of context characteristics required for a particular user task. A road monitoring application such as [MPR08], for example, may be fine-tuned to support the recognition of potholes or honking cars to detect a crowded intersection. Similarly, an application to estimate the travel time may be fine-tuned to estimate a car's speed on the basis of its acceleration [TRL+09]. A social networking application, on the other hand, such as [MLEC07] may be able to identify user activities to generate status updates on social networking platforms such as Facebook [Fac] and MySpace [Mys]. These examples demonstrate that the existing systems and applications target a variety of context types and there is a need to have cohesion between the systems and applications to enable a seamless and resource efficient experience for the both, the developers and the users.

Outside of a laboratory environment, however, users are often involved in multiple tasks at a time which requires a simultaneous execution of several applications. As a simple example consider that when driving around a user might want to use both, the travel time as well as the road monitoring application, while using the social networking application to update the status with the music playing on the radio. As a result, the applications have to continuously sample and process acceleration data for road monitoring and speed estimation as well as audio data for music classification and honk detection. Yet, since running multiple applications simultaneously increases the energy requirements additively and due to the fact that personal mobile devices are usually battery powered, this approach is inherently limited in scale. This highlights need for energy efficiency

mechanisms such that the execution of multiple context recognition applications on these resource constrained devices does not impact their usage for other purposes, such as making calls, browsing internet etc.

2.2 Contribution

The thesis provides two contributions to the field of ubiquitous computing in general and context recognition using personal mobile devices in particular. The contributions are housed together as a single generic and energy efficient context recognition framework for personal mobile devices, particularly smart phones. By generic we mean that the framework should be able to provide means for the creation and execution of applications targeting any type of context. By efficient we mean that the framework should provide mechanisms to enable energy efficient execution of context recognition applications.

The first contribution of this thesis is the provision of a generic context recognition framework for smart phones. In order to realize this contribution, the framework provides different abstractions, supports runtime execution and provides off-line development tools to develop context recognition applications for smart phones. At a high level, the framework consists of two systems namely the component system and the activation system. The component system uses a component abstraction and uses a component model which consists of component, connectors and configurations for the development of context recognition logic independent from the target context. A context recognition logic is realized through configurations. A configuration is a set of components connected using the connectors. The runtime system of the component system uses the configurations to instantiate the components and the associated connectors. The component system is supported by off-line development tools to enable rapid prototyping of the context recognition applications. These development tools include a component tool-kit which consists of a large number of components belonging to the three logical levels of a typical context recognition application namely, the sensing level, the preprocessing level and the classification level. The off-line tools also include a graphical editor implemented as an Eclipse [Ecla] plug-in to enable developers to create configurations by simply dragging, dropping and connecting the components. The graphical editor also provides configuration validation and code generation utilities. It will be shown in the coming chapters that depending on the number of components in a configuration, the amount of code to represent it could be very large. Therefore configuration validation and code generation mechanisms prevent developers from committing coding related errors.

The activation system uses state machine abstraction to enable task dependent determination of context information. The state machine model used by the activation system consists of states and transitions. A state consists of configuration(s) developed using the component system and describes a particular step in the overall context recognition logic where as a transition describes switching between the steps. The configuration(s) associated with a particular state can represent high level context information such as recognition of a particular activity by a user in a day or it can represent low level context information

which can be part of some higher level context information. In any case, the states are connected through transitions which are modelled using a rule based abstraction. A typical rule to represent a transition consists of three parts namely, the operand, operator and the value. The operand represents the context recognition logic, the operator represents some mathematical function and the value represents a threshold. In order for a transition between states to occur, the rule associated with the transition has to be evaluated to be true. The runtime system of the activation system instantiate the state machines by collaborating with the component system and uses a rule engine to evaluate the transitions. The activation system is supported by off-line development tool to enable rapid prototyping of the state machines. The development tool includes a graphical editor implemented as an Eclipse plug-in. Similar to the editor for the component system, it also enables developers to create, validate and generate source code for the state machines. The state machine abstraction used in the activation systems also enable generic applicability of four energy efficiency techniques namely Suppression, Substitution, Adaptation and Piggybacking as described in [ZKS10].

The second contribution of this thesis is our novel energy efficiency technique called configuration folding. Configuration folding enables energy efficient execution of simultaneously executing context recognition applications on smart phones by identifying and removing the functional redundancies between them. As an output, the configuration folding produces a single redundancy free set of functionalities such that this set can fulfil functional requirements of all the applications. The redundancy free set of functionalities ensures that using the same set for all the applications does not have any impact on the functional correctness of the output of the individual applications. This means that if applications are executed with and without configuration folding, their output remains the same. We also investigate the impact of parametrization differences on configuration folding. Towards this end, we identify commonly used parameters in context recognition applications, classify them and identify relations between their values. Consequently, we provide an extended version of configuration folding.

The thesis also discusses the evaluation of the context recognition framework and the associated configuration folding technique. The framework has been extensively used in the development of different context recognition applications and prototypes in different European Commission projects such as GAMBAS [gam] [AIP14], Living++ [liv] [IFW+13], SmartKye [sma] and BESOS [bes]. The thesis discusses the different applications that have been developed and also highlights the advantages provided by the associated off-line development tool support. The thesis also evaluates configuration folding by creating real world applications using the component system and analysing the energy savings for them when configuration folding is applied. The experimental evaluation of configuration folding without parametrization support shows energy savings between 13 and 48% when applied to music and speech recognition applications whereas the energy measurements of configuration folding with parametrization support shows energy savings of up to 45% for the test scenarios used in the experiments.

The major contributions of this thesis namely the component system, the activation system and the configuration folding have been published in different scientific conferences namely SUTC 2010 [HIA+10], Caseman 2010 [HIAM10], PerCom 2012 [IHW+12b] [IHW+12a], HomeSys [IFW+13] and IE 2015[IHM15]. Apart from the publications in the conferences and workshops, the framework has been used in student projects and theses done at the University of Duisburg-Essen, Germany.

2.3 Structure

The rest of the thesis is structured as follows. In the next chapter, we provide concepts and definitions to facilitate the understanding of work done in this thesis. In Chapter 4, we provide a detailed discussion of the existing context recognition approaches and identifies their shortcoming. In Chapter 5, we give an overview of our framework by discussing the design requirements, the design rationale behind the framework and a description of different building blocks of framework's architecture. In Chapter 6, we discuss the component system in detail. We describe the component abstraction, the development tools and provide the evaluation of the component system. In Chapter 7, we discuss the activation system. We describe the state machine abstraction, the development tools and also provide the evaluation of the activation system. In Chapter 8, we discuss configuration folding and its evaluation. In Chapter 9, we conclude the thesis and provide an outlook on the future work.

3 PRELIMINARIES

In this chapter we discuss preliminary concepts and definitions that are required for the understanding of the contributions provided by the thesis. We start by defining ubiquitous computing with three motivating scenarios to highlight different concepts that are used in this thesis. These concepts include definition of context, platforms for context recognition and software aspects of these platforms. In addition the chapter also defines some tools that have been used during the course of the thesis.

3.1 Ubiquitous Computing

Mark Weiser in his seminal paper "The Computer for the 21st Century" [Wei99] presented the idea for modern day computing. This idea resulted in the foundation of a new branch of computing called ubiquitous computing.

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it" – (from "The Computer for the 21st Century" by Mark Weiser)

Ubiquitous computing envisions applications which provide seamless and distraction-free support for the everyday tasks of their users. In order to realize this vision, applications must be able to adapt to the dynamics of the surrounding environments and to the varying user requirements. The adaptation must be performed automatically in order to ensure that it does not conflict with the goal of providing a distraction-free user experience. This in turn requires applications to consider different characteristics of user's context at runtime. Examples of the context characteristics may include user's current location and user's activities. Next, we discuss three scenarios to define the meaning of context used in this thesis.

3.2 Scenarios

We use three scenarios to define the concept of context. These scenarios highlight the need for information associated with different aspects of a user. These different information aspects when put together can provide an estimation of user's context.

3.2.1 Assisted Car Parking

Consider that a user has to receive a guest at the airport. He notes down the appointment in his on-line calendar with details such as the time of guest's arrival and location of arrival gate. One hour before the arrival, the user picks his smart phone and leaves the office. As the user leaves, the mode of transportation recognition application on the smart phone detects that user is walking out of office and checks his online calendar to determine his next location, which is the airport. As the user start driving, the same application recognizes that the user is now driving towards the airport based on his speed of movement, noise from vehicle's engine and noise from the other vehicles. As the user arrives at the airport the parking finder application communicates with the parking service of the airport and navigates user to a free parking spot close to the arrival gate.

3.2.2 Assisted Shopping

Consider a user wants to go shopping. The user creates a shopping list and save it on his smart phone. As the user leaves the house and walk towards his car, the smart phone recognizes changes in location from indoor to outdoor and determines that user is going for shopping. As the shopping center is close to user's residence and there is still adequate time left before the shopping center closes, the shopping assistant application on the smart phone recommends the user to walk instead of drive. As the user enters the shopping center, the application recognizes it based on the address of the shopping center, audio announcements made in the center and sounds of other people shopping. The application then communicates with shopping service of the shopping center and arranges the list of items on shopping list based on their vicinity to the user. The application then prompts user when he is close to a particular shopping item and navigates him towards it.

3.2.3 Assisted Living

Consider that an elderly person lives alone. In order to keep track of his well being, his relatives have installed different sensors in his residence which keep track of different parameters such as his location (kitchen, bedroom etc.), his movement, his medication intake, his social interactions etc. In case of any anomaly the system communicates with the elderly person to remind him for the things he is forgetting. In addition, the elderly person also wears health monitoring devices on his body to measure vital parameters such as heart rate. The sensors on on-body devices report their readings to a remote processing unit which process the raw values and compare them against thresholds set by the relatives of the elderly person. In case any of these parameters crosses a certain threshold, the relatives or the emergency services are notified through a phone call or a SMS message.

3.3 Context

Based on the example scenarios, the context of a user or an entity in general can be described as a function of entity's environment and the activity in which the entity is involved. In this thesis an entity refers to a human user and we use these terms interchangeably. Specifically, we define context as the following.

Context of entity is defined as the environment in which the entity is present and/or the activity in which the entity is involved in. The environment of an entity is characterized by its location, physical characteristics of the location and activities of other entities in that location where as the activity of an entity is the physical work the entity is performing.

Previous works such as [SAW94] and [SBG99] used entity's location and entity's activity as the definition of context respectively. We argue that the context of an entity may require information about the location as well as the activity. The entity's location can be generalized to an environment in which the entity is present. An environment not only consists of location but also takes into account the activities of other entities in that location as well as the physical characteristic of location such as light intensity, background noise etc. The activity of an entity is any physical work such as climbing up stairs, using a computer, driving a vehicle, talking to a friend etc. The examples scenarios also support this definition. In the assisted car parking example, the information about user's presence in the car and presence at the airport represents the user's environment which is identified using the location of the airport, noise of planes landing and taking-off where as the information about user's driving towards airport and greeting and meeting the guest represents user's activity. In the assisted shopping example, information about user's presence at home and at the shopping center represents the environment whereas the information about user doing shopping represents the activity. Similarly, in the assisted living example, the environment of user is represented by his presence in the living room, kitchen etc, whereas his movement, medication intake, social interaction represents activities.

3.3.1 Context Recognition Application

The example scenarios presented previously mentioned use of mode of transportation recognition, parking finder and shopping assistant applications. These applications provide assistance to the user in carrying out the respective tasks. In order to provide the assistance, the applications are required to understand the context of the user. The mode of transportation application is required to understand when the user is in a vehicle and when the user is on foot. Similarly, the parking finder application must identify that the user is in the parking lot and how a best parking place can be found for him. The shopping assistant application must understand that the user is going for shopping and how his shopping experience can be made as distraction free as possible. All of these applications have one

requirement in common i.e. ability to understand the context of the user. These applications understand the context by gathering information from different sensing sources, process them and classify the outcome. The sensing sources could be local to the platform on which these applications run or could be distributed in the environment. The processing of data coming from these sensing sources involves extracting of features which are used to classify the context. We call this process context recognition and thereby, applications executing this process are called context recognition applications.

The context recognition applications may differ greatly from each other. The reasons for the difference could be many. The difference could be in the number of devices, heterogeneity of devices, and number of context sources and heterogeneity of these sources. For the assisted car parking example above, the user relies on his smart phone for determining when he is driving. In this case, the user uses a single device for determining his context. In the assisted living example, the elderly person uses smart phones, wearable devices and other sensors installed in his residence to determine his context. In this case the determination of context has higher complexity because it involves multiple and heterogeneous devices and heterogeneous sources of data which results in bigger challenges for determining the context compared to when a single device is used.

3.3.2 Context Recognition Process

A typical context recognition application follows three steps for determining the desired context. These steps include sensing, preprocessing and classification. Typically these steps are followed in bottom-up fashion i.e. from sensing to classification. However, there also exist systems which do not follow this flow strictly and use feed-back mechanisms to enable specific requirements such as not performing preprocessing if the sensing phase does not return usable sensor data, for instance. In the following we take a brief look at each of the three steps involved in a typical context recognition process.

Sensing

Sensing [LML+10] refers to the gathering of raw sensor data from the sensors either installed on a local device or on a remote device. It is the first step in context recognition process. Sensing sources can be of two types namely the physical sources and virtual sources. The physical sensing sources are the ones which are physically present on the device where as virtual sensing information sources are those which do not sense the data directly but instead the data is stored on them either by a user or some external process. Examples of physical sensing sources are microphone and light sensors where as examples of virtual sensing sources may include on-line calendars, to-do lists etc. In the following we describe some of the most commonly used physical sensors that are typically found in many existing context recognition devices.

- Audio sensor: This sensor captures audio data. The audio data could represent human voices, ambient sounds such as the music played in the background, noise from the crowd in a stadium, noise from cars in traffic congestion on a road.

- Accelerometer sensor: This sensor captures motion. The motion data can represent different movement modalities such as standing, walking, running, driving a car, travelling in a bus or a train etc. The most commonly used accelerometers for context recognition purposes is 3-d axis accelerometer which reports measure of force on the three axis namely the x,y and z axis. Typically, the accelerometer data is aggregated over the three axes to measure the magnitude so that the effect of orientation change is discarded.

- WiFi sensor: This sensor scans the surroundings to gather list of visible WiFi access points. The access points are normally identified using a unique identifier (SSID), MAC address and the relative signal strength indicator (RSSI) value. Using this information, different techniques such as finger printing can be used to determine the location of the user or the location of the device.

- GPS sensor: This sensor provides Global Positioning System (GPS) coordinates which tells the current location of the device or the user of the device. Using the location information from GPS sensors, the context of the user can be determined. In contrast to WiFi data which can provide estimate of indoor location, GPS data is more accurate for the outdoor locations.

- Light sensor: This sensor measures the intensity of light which can give a clue about the surrounding environment e.g. indoor, outdoor etc.

- Temperature sensor: This sensor measures the temperature of the surroundings. The temperature information can be used in assisted living scenarios to determine the safety of the user e.g. whether the temperature of the room in which the user is present is too high or too low.

- Orientation sensor: This sensor provides information about the direction of the device or the user carrying it. If the context recognition approach is dependent on the orientation, then data from this sensor can be used to adapt the recognition approach.

Depending upon the application requirements, multiple of these and other sensors can be used to determine the context. In order to determine whether the user is using a treadmill in a fitness center, multiple sensors would be required e.g. accelerometer would be require to determine movement, WiFi would be required to determine the location of treadmill inside the fitness center, audio sensors would be require to determine the sound of treadmill and other equipment in the vicinity, for instance.

Preprocessing

Preprocessing is the second phase in a typical context recognition process. The data from the sensing phase is fed into different mathematical functions to identify the characteristic of sensing data which could help in determining possible contexts. The preprocessing

process may consist of many functions, performed in parallel or in sequence. The number and sequence of these functions is usually determined by the application requirements. Generally, preprocessing functions are classified in two categories, time domain and frequency domain. The time domain preprocessing functions operate on the sensor data as it is without applying any frequency transformations. Examples of time domain functions include statistical functions such as calculation of minimum value, maximum value, mean value etc. It also include other functions such as zero crossing rate, low energy frame rate, windowing etc. The frequency domain functions are needed when time domain functions alone are not sufficient to provide conclusive information. The conversion of time domain sensing data to frequency domain data typically involves application of certain frequency transforms such as Fast Fourier Transform (FFT). In this thesis, we have developed different context recognition applications which use different preprocessing functions. In the following, we describe the preprocessing functions that we have used for our speech and music recognition applications as examples.

- Low Energy Frame Rate: The low energy frame rate [SS97] identifies number of subframes within a frame which have root mean square values 50% or less than the root means square value of the frame. In human speech, as there are more silent pauses than in music, therefore, the low energy frame rate for speech is usually higher than that of music.

- Zero Crossing Rate: The zero crossing rate [SS97] determines the number of zero crossings in a frame of audio data. It is determined by counting the number of changes in the sign of the signal between positive and negative. Usually the human speech consists of voice and has more pauses than music, therefore, the variation in zero crossing rate for speech is mostly higher than that of music.

- Fast Fourier Transform: The Fast Fourier Transform [Wal96] transforms the time domain data into frequency domain data. The reason for frequency domain analysis is that the frequency components of a signal can exhibit characteristics which are not visible with the time domain analysis. The Fast Fourier Transform is basically a discrete fourier transform with the runtime complexity of O(nlogn). The input of this transform is a discrete periodic time domain data and the output is a discrete frequency domain data. In order to make the input and output data periodic, different time domain windows are applied on the input data. In other words, only a fixed size frame is sent as an input and a fixed size frame comes out as an output.

- Spectral Centroid: The spectral centroid [LSDM01] determines the balancing point in power spectrum of input frame. The human speech has fewer frequency sounds, therefore the spectral mean of speech frames is usually lower than the spectral mean of music frames.

- Bandwidth: The bandwidth [LSDM01] determines the range of frequencies present in an audio frame. The human speech usually has fewer frequencies than music therefore bandwidth for music is usually higher than bandwidth for speech.

- Spectral Rolloff: The spectral rolloff [LSDM01] determines the frequency component below which a certain concentration of frequency distribution is present. The amount of concentration of frequency distribution below the frequency component is usually 92 or 93 % of the whole distribution. The human speech usually has low value of spectral rolloff compared to music.

- Spectral Entropy: The spectral entropy [LPL+09] determines the amount of useful information present in the input frame. Lower entropy means more useful information where as high entropy means less useful information e.g. an audio frame with silence has a flat spectrum which means no or less useful frequency characteristics whereas audio frame with few frequency components means more useful information. Entropy is calculated by normalizing the spectrum of an input frame where as the spectrum is obtained by taking the frequency domain transform of the frame.

- Spectral Flux: The spectral flux [SS97] calculates the difference in the power spectrum of consecutive frames. The difference is calculated using Euclidean distance. For human speech the difference in power spectrum of consecutive frame is usually higher than music because of more pauses in the speech. However, for certain music the same behaviour can be observed.

Classification

The classification step takes the output of preprocessing step (feature vector) and determines the context from the list of possible contexts that can be detected with the computed feature vector. The classification step involves use of classification algorithms and these algorithms require learning before they can be used for classification. The learning can be supervised and unsupervised. In supervised learning, the classification algorithm is provided with a set of labelled data in the learning phase whereas in unsupervised learning, the classification algorithm is provided with unlabelled data and the algorithm treats each sample of training data independently to create a classification model. Next, we take a brief look at some of the classification algorithms used in context recognition applications.

- Decision Tree: The decision trees [Lan91] are a tree based structure in which the classification category lies at the leaf nodes. The non leaf nodes represent different features of the computed feature vector and branches between nodes represent transitions from one node to another. The structure of a decision tree is created using a learning algorithm. Most commonly used algorithms are ID3 [Qui86] and C4.5 [Qui93]. Decision trees are computationally inexpensive when the tree size is small. However, the learning process is time consuming and consequently the retraining of classifier to accommodate more contexts as well. We have used decision tree classifiers for all the context recognition applications that we have developed for this thesis.

- Neural Networks: The neural network [MSTC94] classifiers emulate information processing of human brain. As a brain consists of billion of neurons which send, receive

and process information from other neurons, the neural network also consist of processing units called neurons. A typical neuron in the neural network aggregates the information that it receives from other neurons and by comparing the aggregated information with a threshold, it sends a positive or a negative output. The inputs to the neurons are weighted which are adjusted for optimizing the classification during the training phase. A neural network typically consists of input, hidden states and output stages. These states consist of neurons such that the output of every neuron in every state is fed to the input of all neurons in the next stage. The hidden state may consist of many stages. However, only the outputs from the neurons in the output states are used as the final result. The neural networks are typically classified as feed-forwards networks or feedback networks. In the former, the information is processed in one direction only while in the latter the information is fed back for optimizations.

- Clustering: The general idea of clustering is to group the preprocessed data which has similar characteristics such that different groups represent different contexts. K-Nearest Neighbour (KNN) clustering and K-mean clustering algorithms [WKRQ+08] are amongst the commonly used clustering algorithms. In KNN clustering, the preprocessed data belonging to different context is grouped based on the distance such as the Euclidean distance. The K-mean algorithm on the other hand classifies a preprocessed data in to a cluster which has mean value closet to the value of preprocessed data.

3.3.3 Context Recognition Platform

A context recognition platform is a microcomputer or an embedded system which provides access to local and/or remote sensing sources. A context recognition platform possesses computational capacity to execute the context recognition applications installed on them. The computational capacity of these platforms is defined by their hardware and software specifications. As a result these platforms may vary and thereby supports execution of only those context recognition applications which are developed for them.

Smart phones

Smart phones are currently the most commonly used platforms for performing personal context recognition. Modern day smart phones are equipped with computationally powerful processors, large memory and a set of large number of sensors to collect the raw data. Typically, modern day smart phones are equipped with audio sensor which provides audio data such speech and voice, accelerometer sensor provides movement data, barometer sensor provides pressure data, WiFi/GPS/GMS sensors provide location data, light sensors provide data about intensity of light, humidity sensors provide data about humidity levels in the air, gyroscope sensor provides data about the orientation etc. The context recognition applications use these sensors through the operating system supported by the smart

phones. In addition to sensors and computational capabilities the smart phones have interactive screens which enable users to interact with local and remote context recognition services.

Tablets

Recently, there has been an increased use in tablets as a personal mobile device in addition to smart phones. A typical tablet also possesses high computational capabilities like a smart phone with few differences such as increased screen size. The tablets though are used mainly for browsing the internet and as e-readers, they can act as an additional source of user's context information. Together with smart phone, recognition of context can be improved e.g. the tablet and smart phone can share locally gathered context information to make finer context classifications.

Smart watches

Similar to tablets, smart watches are increasingly becoming technically advanced with support from different sensors such as microphone, accelerometer, Bluetooth etc. Consequently, they can also act as an additional context recognition source. Though the smart watches do not possess computational capabilities similar to tablets or smart phones, they however can be used to perform computationally less expensive context recognition. Similar to tablets, they can interact with smart phones and together these three platforms can improve the quality of gathered context information of the user.

3.3.4 Operating Systems for Context Recognition Platforms

The two most commonly used operating systems for smart phones include Android [anda] and iOS [App]. The context recognition framework presented in this thesis is developed for Android based smart phones.

Android

Android [anda] is a Linux based open source operating system for personal mobile devices from Google. Android provides an application development framework which developers can use to create context recognition applications for Android platforms. The application development framework provides APIs which allow access to local and remote sensing sources. The applications are developed using Java language. The basic concepts required for developing applications using Android include Activity, Service and Intents. Activity is the screen shown to user through which a user can interact with the application, enter data and see the response. Service is something which runs in the background and its main purpose is to do work which does not require visualisation and may also involve lengthy operations such as downloading of a file from a server. Intent is the message passing mechanism through which different components of application such as activity and

activity or activity and service or service and service communicate with each other. Next, we briefly describe the important components for Android application development.

- Activity: An activity [Andb] is the content shown on the screen of an Android based phone, tablet or watch. An activity has a life cycle managed through different system calls. Each call takes application to a particular state. There are altogether 6 states in activity's life cycle. These include Created, Started, Resumed, Paused, Stopped and Destroyed state. A state is entered through a method call. The method call for Created state is onCreate(). The onCreate() method is used for initializing objects and variables when the activity is first created. The method call for Started state is onStart(). The onStart() is called after the onCreate() call, when the activity is created first and it is called every time when the activity enters from Stopped state to Started state. The onResume() method is called before entering Resumed state. This method is usually used for registering receivers for receiving intents sent from other application components. In the Resumed state, activity is visible to the user and is ready for interactions. In case another activity is started on top of the exiting activity, the previous activity enters in the Paused states if it is partially visible or in the Stopped state if it completely invisible. The method calls for Paused and Stopped state are onPaused() and on Stopped() respectively. These methods are use for saving the state of objects and variables used by the activity so they can be restored when activity is resumed again. They are also used for un-registering receivers. When the activity enters in the Destroyed state through onDestroy() call, it cannot be resumed.

- Service: A service [Ande] is used to perform work of long durations which does not requires user interaction such as downloading a file from a server or sampling data from the sensors. A service runs in the main thread of the application which means it is recommended to start the work of a service in a separate thread, otherwise if the service runs in the main thread and blocks it for more than 5 seconds the Android application will stop responding and consequently stop working. There are two types of services namely the Started service and the Bound service. The Started service is not connected to the application component which starts it and once started it will continue even the starting component is stopped. A Started service must be stopped explicitly otherwise would result in leakage of resources. A Started service is typically used when interaction between the starting component and service is not required. The second type of service is Bound services. This service enables interaction between the service and the component which started it. Multiple components can be bound to a Bound service. A Bound service is executed as long as there is at least one component connected to it.

- Intent: Intent [Andc] is the message passing mechanism in Android applications. Intent consists of intent filter which is used as a unique identifier for the application component from which the message originates. Other application components

who want to receive this message must register receivers with this identifier. A receiver is an application component which is used to receive these message through its onReceive() method call. Intents can be registered programitically or through AndroidManifest file. If intents are registered programitically then the components in which they are registered can only respond if they are executing when the intent is sent. If the component is not running then it cannot receive the intent messages. If the intents are registered in the AndroidManifest file, then it is not necessary for the receiving components to be executing. In addition to message passing, intents are also used for starting other application components such as other activities and services.

- AndroidManifest: AndroidManifest [Andd] is a XML based description of an Android application. It contains necessary information required to execute an Android application. This information includes names of all the activities, services and intent filters in the application. It also describes all the security permissions that the application requires to function. If any of this information is missing, Android will not start the application and even it starts it the application might crash when a component is found missing in the manifest file.

- Layout: The user interface in Android applications is described through XML based layout files. The layout files contain description of different visual components such as buttons, text views, background images etc. These layouts are attached to an activity when the application starts. Same layout can be used for different activities and each activity can have its own layout.

- Parcelable: It is the Android's way of performing marshalling and de-marshalling of objects between different components. The objects are passed as Parcels. In order to create a Parcelable, the Java class must implement the Parcelable interface. A Parcelable object can be passed between different application components through intents.

iOS

The iOS [App] is the operating system used in mobile devices developed by Apple. The iOS based devices are also one of the most widely used devices in the world. These devices include iPads, iPhone, iPod etc. The development language for iOS is Objective-C. In iOS the equivalent component of Activities in Android is called UIViewControllers. Similar to Activities in Android, the UIViewController also has a life cycle controlled through different method calls. The init(coder:) method is called just before the content is shown on the screen. The method is usually used for defining view or subviews and attaching them to the main View. Similar to onCreate() method, the init(coder:) method is also called once. The viewDidLoad() method is also called once for performing initialization of views and subviews. The viewWillAppear() method is called after viewDidLoad() just before the appearance of a view on the screen. Unlike, init(coder:) and viewDidLoad(), this

method is called every time a view is appeared on the screen so it can be used to perform functions such as network management, refreshing screen etc. The viewWillDisappear() method is called just before a view is closed and is used for storing the state of the view and other objects and variables used in the view. In iOS, there is no equivalent component for Services in Android. However, by specifying certain permissions to the applications, certain tasks can be executed in the background. There are three types of tasks which can request such permissions. They include playing of audio data in the background, updating of locations information and tasks which require periodic updates from external applications or sources. The iOS App Extensions is somewhat similar to the Intents in Android and lets applications share their contents with other applications. With App Extension, one application provides its extension which can be used by other applications when they use that extension e.g. if an application has an App Extension which reports data from stock exchange then another application can use this extension to show the stock data information to its users.

3.3.5 Miscellaneous

In this section we discuss miscellaneous development tools that we have used to develop off-line tools support for our framework.

Eclipse

Eclipse [Ecla] is an integrated development environment (IDE) for creating software. We have used it in this thesis for creating all systems and applications. Eclipse is a plug-in based architecture and allows easy customization depending upon developer requirements. We used Eclipse java development tools (JDT) for creating all the software for this thesis. We also used Android development tool (ADT) plug-in for creating and building Android based applications. Apart from JDT and ADT we also used Eclipse Modelling framework and Eclipse Graphical editing framework for the development of graphical editors for the systems developed in this thesis.

Eclipse Modelling Framework

Eclipse Modelling framework (EMF) [Eclb] is a data modelling tool for creating applications and editors. The EMF has a code generation utility which can be used to generate Java code for the data model. The EMF model is created using the Ecore format which is similar to Unified Modelling Framework (UMF) models. A model entity in Ecore format consists of Epackage to describe the namespace, EClass to describe the name of the entity, EAttribute to specify the attributes of the entity and EReference to specify the relation of this model entity with other model entities. When code is generated for the model the Epackage is converted to a Java package, EClass to a Java class, EAttribute to Java primitive class member variables and EReference to class variables indicating is-a, has-a etc. relationships.

Eclipse Graphical Editing Framework

Eclipse Graphical Editing Framework (GEF) [Eclc] allows creation of graphical editors for the EMF models. The Draw2d [Dra] framework based on standard widget toolkit (SWT) is used as the basis for creating these editors. The GEF provides different set of classes to enable their creation. In the following we provide a brief description of some of the most fundamental classes.

- EditPart: The EditParts provide mapping of the model elements to their graphical representations. Every model element which requires a graphical representation must have an associated EditPart. There are two main EditParts namely, the GraphicEditPart and the ConnectionEditPart. The GraphicEditParts are used for representing model elements where as the ConnectionEditParts are used for representing connections between the model elements. The GraphicEditParts are drawn using Figures. The Figures are the graphics that represent a GrahicEditPart and therefore the GraphicEditPart must provide implementation for its visualization i.e. how does a Figure look e.g. a square box with round edges, its creation and its refreshing. The ConnectionEditParts are represented by connections. The connections connect different GraphicEditPart. In order to connect GraphicEditParts, the GraphicEditParts must provide ConnectionAnchors. The ConnectionEditParts then connect different GraphicEditParts through these ConnectionAnchors.

- Request: The Request provides means for communication between EditParts and between EditParts and user input. An example of Request could be a user input for creating a EditPart. The EditPart for which a Request is made passes the Request to EditPolicy. The EditPolicy then creates a command for responding to that Request. There are three types of Requests namely the CreateRequest (for creating an object of a model elemtent), GroupRequest (for requesting certain actions to be taken on a group of model objects such as resizing them) and LocationRequest(for enabling specific function for specific part of the EditPart).

- EditPolicy: The EditPolicies allow editing of EditParts. Without an EditPolicy the EditPart is almost of no use. An EditPart can have multiple EditPolicies. Each EditPolicy is assigned a role. The roles are used for categorizing the type of response the EditPolicy can give to a Request. An example of a role is a DirectEdit role. When an EditPart has a policy with DirectEdit role and when the EditPart is double clicked, it becomes editable and the user can change the exposed properties of the EditPart, e.g. it can change the label of the EditPart to maintain uniqueness in case there are multiple instances of the same EditParts drawn on the screen.

RapidMiner

The RapidMiner [rap] is a data mining tool. It provides implementation of different machine learning algorithms. It provides a graphical editor which can be used to create data

flow diagrams with data elements necessary to create classifiers and measure their performance. A simple data flow diagram may consist of an input data model element whose output is connected to a classifier model. The input data model contains the data required for training the classifiers. The data can be in different formats such as .csv or .xcel. The classifier model consists of the type of the classifier to be generated e.g. a decision tree classifier. We have used RapidMiner for the creation of all decision tree classifiers used in the applications developed in this thesis.

3.4 Summary

This chapter has described different concepts and tools used in this thesis. The chapter has provided three example scenarios to define context. The chapter has described the typical context recognition process and described different sensing, preprocessing and classification functions. The chapter has discussed various context recognition platforms. The chapter also gave details about the basic concepts used in Android based application development. Towards the end, the chapter gave a brief overview of Eclipse modelling framework, Eclipse graphical editing framework and RapidMiner tool used in the thesis.

4 RELATED WORK

In this chapter we provide an overview of the existing work related to the contributions provided by the thesis. The discussion on the related work is divided into three parts. In the first part, we discuss various existing context recognition applications based on personal mobile devices such as smart phones to demonstrate their wide spread use. Next, we discuss existing context recognition frameworks which help in creating such applications. In the later part of the discussion, we focus on the energy aspects of running context recognition applications on these resource constrained devices. Based on the existing work on the context recognition frameworks and energy efficiency techniques, we identify the gaps which are covered by the contributions made by the context recognition framework presented in this thesis.

4.1 Context Recognition Applications

There exist a number of context recognition applications for personal mobile devices such as smart phones targeting various domains. These include applications for determination of sound based, motion based, location based and activity based contexts. The type of context determined by such applications could include determination of type of environment in which a user is present based on the ambient sound e.g. whether a user is in an office or at home or at a party, the type of vehicle the user is using based on the motion e.g. whether the user is driving a car or standing in a bus, the type of activity the user is performing based on the location e.g. if a user is in a shopping mall then he is most likely shopping and if he is in a office he is most likely to be working etc. These broad ranges of contexts indicate need for a large set of applications with different requirements. Consequently, a number of context recognition applications such as [EML+07], [HTGT13], [LPL+09], [LCB06], [Bar04b] for personal mobile devices especially for smart phones and others have been developed. In the following we discuss some of these and other applications in detail.

Nericell

Nericell [MPR08] uses accelerometers, microphones and GPS/GSM sensors of smart phones to detect uneven road conditions. The idea behind Nericell is to use these sensors to gather information about the potholes, bumps, braking and honking by using the smart phones carried by the driver or the passenger. Nericell achieves this by performing virtual reorientation of the phone's accelerometer so that the irrespective of the phones position,

the gathered information is accurate and is not affected by the device's actual orientation. Nericell uses phone's microphone to listen the honking to determine traffic congestions and chaotic traffic conditions. The data gathered for the road and traffic condition is transferred to the Nericell server ensuring user's privacy. Thus, we can see that Nericell requires determination of different contexts related to driving i.e. whether user is driving on a bumpy road or a road with potholes, for instance.

VTrack

VTrack [TRL+09] is a travel time estimation system which allows users to avoid traffic jams and road congestions. VTrack achieves this by using GPS and WiFi sensors present on the user's smart phones and with the help of hidden Markov model based map matching and time estimation techniques, it computes the travel time on the route on which the users have driven. The estimated travel times for road segments are communicated to the other users driving on the same road segments. A similar system is proposed by [TBGE10]. This system provides real time transit tracking support to its users by identifying whether the user is travelling in a vehicle, if so then if it is a transit vehicle such as a bus or underground vehicle such as metro and what is the route of the vehicle. Similarly, Madrid Navigator [gam] is another application for smart phones which provides real time tracking and navigation support for the EMT [emt] bus network in Madrid. The Madrid navigator uses the already available web services on the EMT buses and presents this information which includes next stop name, destination stop name and news on any incidents on the route to the user. In order to preserve energy consumption the application uses GPS when it is absolutely necessary, for example to navigate a user through a walking segment to the bus stop or his/her destination.

SoundSense

SoundSense [LPL+09] is an audio based system for smart phones which provides classification between music and speech. The classification is performed by employing combination of supervised and unsupervised learning algorithms. The system uses a number of time and frequency domain pre-processing functions. Some of these functions are low energy frame rate, zero crossing rate, spectral flux, spectral entropy, bandwidth etc. Using the output of these preprocessing functions a decision tree classifier and a Markov model classifier is used for fine grained classification. In addition to supervised learning, SoundSense also use unsupervised learning techniques to classify ambient sounds. There are many other systems which use sound and voice as an input to provide speech recognition [RJ93], speaker identification [Rey02] and music genre identification [TC02].

iFall

iFall [ST09] is a fall detection and alarm triggering application for Android [anda] based smart phones. iFall uses changes in phone's accelerometer values to determine falls. It does so by continuously sampling the accelerometer readings and when there is a sudden

rise of acceleration values (above a upper threshold) immediately after a sudden drop of acceleration (below a lower threshold), the application consider it as a possible incident of fall and wait for certain time to find any change in acceleration for avoiding any false positive before signalling detection of fall. Fall detection using accelerometer has also been studied in other works such as [ARK+06] and [BvdVG+10], for instance.

Transport Mode Detection Applications

[HNT13] provides mode of transport detection system using accelerometer of a smart phone. Authors use a set of classifiers for identifying the transportation mode. A kinematic motion classifier is used to perform coarse classification between pedestrian and other kinematic modalities. If the kinematic classifier does not provide any useful results, the system uses a stationary classifier to determine whether the user is in a motorised transport or not. In case the user is in a motorised transport then the motorised classifier is used to identify whether the user is in a bus, train, metro, tram or car. There exist other techniques such as use of speed and location by using GPS to determine the mode of transportation [ZCL+10] but using GPS is energy consuming and also does not provide good coverage in crowded urban centres and undergrounds. In order to mitigate this, some other solutions such as [RMB+10] use combination of GPS and accelerometer. Another alternate approach that has been presented in [SVL+06] is to use the GSM cell tower information to identify mode of transportation by measuring the rate of change of cell towers the user is connected to. [MEBH08] used combination of WiFi and GSM to determine mode of transportation. [SZG+14] demonstrated use of barometers instead of accelerometer for determining user activities such as walking, idle and in vehicle.

Wearable Computing Applications

Apart from solely relying on smart phone for context recognition, there has been significant work such as [WLTS06], [LY13], [HKAK08], [BI04], [EPMK08], [LL13] which uses body worn sensors for determining user context. [WLTS06] uses body worn accelerometer and microphone sensors on user's arm to determine activities such as sawing, hammering, drilling, grinding, sanding, opening a drawer, tightening a vise and turning a screw driver. [HKAK08] uses a Bluetooth tri-axis accelerometer and a RFID based iGrabber (a RFID reader to wear on hand) for determining various activities such as sitting, standing, walking, lying, running, hand shaking, rope jumping, reading+sitting, bush hair+ standing. [BI04] uses five biaxial accelerometers on different body parts of the user to detect various physical activities. The activities were classified using different preprocessing functions such as mean, energy, entropy, correlation and a set of classifiers. Similarly [EPMK08] uses wearable computing devices to gather user fitness information. The gathered services include lying down, sitting, standing, walking, running, cycling, rowing, playing football and Nordic walking etc.

4.2 Context Recognition Frameworks

There exist a number of context recognition frameworks such as CRN Toolbox [BLA08] [BKLA06], BeTelGeuse [NKL+07], Context Toolkit [SDA99], Yale [MWK+06], PCOM[BHSR04], Context Phone [ROPT05], MyExperience[FCC+07], Common Sense Tooolkit[Com] and CenceMe [MLEC07]. The main purpose of these frameworks is to provide developers a platform and a code base which they can use to develop new applications and increase their productivity by reusing the already implemented context recognition logics by other developers. In the following we take a look at some of these frameworks.

Context Toolkit

The context toolkit [SDA99] uses context widgets to provide the context information to the applications. The basic idea of context widget is similar to the idea of widgets in graphical user interfaces (GUIs) i.e. as widgets in GUIs provide an interface between the implementation of the graphical components and the applications using them, similarly, context widgets provide interface between the applications and the actual sensing and preprocessing logic required for determining the context. However, there are important differences between a GUI widget and a context widget. A context widget is deployed in the environment and is active all the time unlike GUI widgets which are controlled by the applications. Also the context widgets are deployed in a distributed fashion and rely on three kinds of components namely the actuators, interpreters and servers. As an example, for an indoor localization application, the localization widgets deployed in an environment would inform the application about any location change when a change take place. The notification mechanism is dependent on the widget and thus could be different between widgets. The context toolkit also enable applications to combine information from multiple widgets by allowing their compositions e.g. if the application is interested in knowing about the activity the people are performing in their classrooms, this could be achieved by an class room activity widget which could be composed of presence widget (to identify people in class room), audio widgets (to analyse the audio) etc.

Context Recognition Network Toolbox

The context recognition network (CRN) toolbox [BKLA06] uses a component abstraction for the creation of context recognition application mainly for wearable devices. The toolkit is written in C++ and is provided with a GUI to enable rapid creation of applications. The main features of the toolbox are parametrizable components for providing application specific accuracy requirements and a runtime engine to manage the data stream from different sensing sources and data flow control between different components. As the sensor data streams originate from different sources, the runtime system also ensures synchronization. The synchronization is performed by searching distinct events in the data streams and storing their time stamp. The synchronization is then performed by adjusting the time stamps of similar events. The data from different streams is then merged by aligning

their time stamps. As an example, acceleration data from two body worn sensors generates an event e.g. high amplitude of acceleration when the user jumps in the air. The runtime system notes the event in the first stream, notes the time stamp and then adjust all the timestamps of later occurrences of this event. The CRN toolbox is equipped with a component toolkit mainly including components for reading input streams, synchronizing streams, filtering, classification and writing streams.

PCOM

PCOM [BHSR04] is a component system for distributed pervasive computing applications. Its main goal is to support automatic reconfiguration of applications in a dynamic environment. PCOM uses stub objects to mediate component interaction across different devices and performs search to determine suitable configurations. The components in PCOM consist of contracts. A contract consists of two parts. One part specify the component's requirements on the executing platform and the second part specifies the functionalities offered by the component as well its dependencies on other components. The PCOM follows a generic adaptation model which share burden of application reconfigurations between users and the developers. With this model, the developers specify the functional and non functional requirements of the services required by the applications and the user is responsible for managing the adaptation performance goal. The communication between devices is supported by BASE [HBS03]. BASE is communication middleware to manage connections between devices in a dynamic environment. Unlike PCOM, the framework targeted by this thesis is aimed at personal smart phones and therefore, the target execution environment is not distributed.

BeTelGeuse

BeTelGeuse [NKL+07] is Bluetooth based data gathering and forwarding tool mainly targeted at wearable computing applications. BeTelGeuse runs on mobile phones and uses the phone's Bluetooth to gather data from different on-body sensors and transfer it to servers for analysis and processing. The core of BeTelGeuse relies on a black-board architecture that can be configured during start up phase, however, at runtime it cannot be adapted. BeTelGeuse supports automatic device discovery by using a device mapping strategy. For this the system relies on Readers. The Readers are used to read data from the sensors. The Readers have unique device identifiers which are used in the device mapping. Upon start-up BeTelGeuse scans for the available devices and only makes a connection with those which have identifiers known to the system.

ContextPhone

ContextPhone [ROPT05] is a software platform which acts as a middleware between the context recognition applications and the operating system functionalities for smart phones. The main design goals of ContextPhone includes (i) provision of context as a resource which means that it is understandable by humans and can be transmitted when requires (ii)

incorporation of already available applications to enable reuse (iii) speedy communication of context information to ensure the sanctity of context (iv) distraction less execution to ensure normal usage of phone and (v) support rapid prototyping i.e support faster inclusion of new data sources, sensors etc. A sample application as presented by the authors is called ContextLogger, which logs all context recognition events to local and remote storages without user intervention for later analysis. The design goals of CotnextPhone highlight important features which should be exhibited by such frameworks in order to ensure their usage by the large number of users.

CeneMe

CenceMe [MLEC07] uses built in microphones, accelerometer and WiFi sensors on smart phones to detect different user statuses. These statuses include user's activity, user's mood, user's habits and user's surroundings. Using the mentioned sensors, CenceMe recognizes the user statuses and injects them to their social networking profiles such as Facebook, MySpace, IM etc. In order to inject the status information, CenceMe requires gathering of user's context information relevant to the status. The activity status in CenceMe can consist of various possibilities i.e. whether the user is sitting, walking or meeting friends. In order to identify activity status, CenceMe uses phones accelerometer to identify user's body posture. Similarly, the habit status also consists of different possibilities such as going to the gym, going to coffee shop, going to work etc., and in order to identify these possibilities, CenceMe determines user location by using WiFi. Thus, we can see that CenceMe requires determination of different contexts to enable more engaging social networking experience for the user.

ToolChains

Tool chains are often necessary to improve the quality of context classification and at the same time to support large scale of users. There has been some work in this regards and some of the examples include [GSBB10] and [Com]. [GSBB10] provides a tool chain for the real time classification of context information. The tool chain consist of a context classification system (CCS), CCS identification algorithms, data collector tool (DCT), context annotator tool (CAT) and context data base (CDB). A fuzzy inference system is used to map the output of CCS identification algorithms to a CSS. DCT facilitates data collection and CAT allows users to annotate the data. The tool chain uses CDB for the efficient management of different streams of data and stores not only the raw data but also the annotated, preprocessed data and trained classifiers. The main functional complexity of the tool chain lies in the CCS identification algorithm subsystem whose task is to read data and apply a subtractive and Gath-Gave clustering followed by least square regression for calculating the fitness of the results. The tool chain also uses a genetic algorithm for optimizing the results. The use of tool chain is very useful in off-line analysis of the context data and with an accurate set of tools it would be possible to provide automated classification of context data in large scale scenarios.

4.3 Energy Efficiency Techniques

Energy efficiency plays an important role in the usage of context recognition applications, particularly for smart phones. A typical context recognition application follows a series of steps. In the first step, it requires processing of sensor data. In the second step the raw data is processed to extract the feature vector. In the last step the feature vector is fed in to a classifier which then determines the context. Various studies such as [KMD13],[PFW11],[CH10], [FGR+12] have investigated the energy consumptions in smart phones. These studies have focused on different hardware components such as motion sensors, audio sensors, location sensors, screens etc. These studies show that certain sensors such as audio sensor and location sensors consume more energy than accelerometer sensors or light sensors. In modern day smart phones, the high resolution screens make them one of the high consumers of phone's battery. Studies show that for obtaining highly accurate context information, a context recognition application typically require continuous or high frequency sampling of sensor data and depending on the sensor the rate of battery depletion could be different. Moreover, if the preprocessing and classification steps also perform computationally expensive operations, the battery is depleted even faster. There has been a considerable work in achieving energy efficiency for context recognition applications using smart phones. Some of the examples include [KLJ+08],[WLA+09],[SBS02],[Nat12],[BP07], [SBCR05], [PHZ12],[JLY+12], [VBH03], [RPKL12], [PLL11], [LZY+12]. In the following we discuss some of them to show different existing approaches.

SeeMon

SeeMon [KLJ+08] is framework for energy efficient execution of applications on smart phones. It achieves energy efficiency by using a bi-directional context recognition approach. Using this approach the framework uses a feedback pipe to push the decision making about context changes from the classification level to the sensing or preprocessing level. As a result, the complex computations for detecting changes in context happens only if there is a significant change in the data observed at the earlier stage of context recognition. As context of a user typically remains constant for certain time, the idea of not processing every data set leads to energy savings. The authors argue that in order to observe changes in context, it is also not necessary to monitor sensor data from all the sensors and a change in context can be signalled by a smaller subset of sensors. Therefore, the framework uses Essential Sensor Set (ESS) which is a group of least number of required sensors to determine if there is a change in context.

EEMSS

Energy Efficient Mobile Sensing System (EEMSS) [WLA+09] is a hierarchical sensor management system which provides energy efficient execution of context recognition applications by executing the sensors in increasing order of energy costs. This means low energy sensors are used first to decide whether use of high energy sensors is necessary. The sys-

tem uses this hierarchical approach to identify different user states such as motion of the user, location of the user and the ambience of the user. The applications uses XML based state descriptors which consist of state name and their associated sensors and conditions. This state descriptor for application is used by the runtime system of EEMSS to execute the applications. Upon entering a state the sensors specified for the state are used in a hierarchical manner to determine certain conditions. Upon fulfilment of conditions, the system shifts to next state description. The authors claim that the system is flexible and determination of new states as per application requirements can be achieved by extending the state descriptor file.

ACE

ACE [Nat12] is an energy efficient middleware for context recognition applications. The middleware uses different approaches to achieve the energy efficiency. These approaches include context inferencing and proxy based speculation in addition to normal context recognition procedures. Context inferencing determines the context based on context information stored in a cache. As an example, if an application asks the system whether the user is in his office but instead of computing context characteristics for being in office, the system finds the current context of driving in the cache, the system then tells the application that user is not in the office. In proxy based speculation, the system uses low energy costing context attributes to determine high energy cost context attributes e.g. if the application ask the system if the user is in office and system does not find any context information in the cache then it computes low energy costing attributes such as if user is jogging, for instance. If result of jogging is turned out to be true, the system tells the application that user is not in the office. In cases when context inferencing and proxy based speculation is not usable, the system computes the context using normal procedure.

SymPhony

SymPhony [JLY+12] is a resource contention management middleware for continuously sensing context recognition applications. SymPhony supports simultaneous execution of multiple applications and efficiently manages resources between them. Symphony introduces concept of frame externalization which means that it try to identify semantically similar data streams between different applications. It uses a frame based coordination and scheduling mechanism. The applications pass their data flow and sensing requirements using a simple API. The data flow is an XML representation of different context recognition functions and connections between them, hence creating a data flow. The sensing requirements from the application specify the monitoring level and the monitoring delay. The monitoring level specifies how often the application requires the context to be monitored where as monitoring delay refers to the maximum tolerable time delay that application can wait for the context information. Using the data flow and sensing requirements from different applications, SymPhony creates scheme for efficient sharing of resource between them.

This discussion shows energy efficiency techniques adopted at the system level such that different applications can benefit from them. There also exist specialized solutions meaning that the applications implicitly include energy efficiency mechanisms in the determination of their context. In the following, we take a brief look at some of such applications related to audio sensing and location based sensing.

Location based Applications

SensLoc [KKES10] is a location sensing system which abstracts location of user in terms of places and paths. SensLoc consists of different building blocks namely the SensLoc Manager for providing the execution and management of location sensing strategies, Path tracker for tracking the user movement, Place detector to determine visited place and the Movement detector to determine user's movement. As place detection is energy consuming process which may require use of GPS or WiFi, its usage is controlled based on feedback from the Movement detector. Hence, the energy efficiency is achieved when user arrives at a particular place as detected by Place detector. The Place detector then asks Movement detector of any movements. If no movement is detected, the Place detector is turned off and is turned on when movement is detected again. A similar approach has been used by [ZKS10] which uses four different energy efficiency techniques for location sensing. These involve (i) using accelerometer to first determine the movement and then perform location sensing (ii) if there are more than one ways to perform localization such that one way is less energy consuming but equal in accuracy to the other way then the one with the low energy consumption is used (iii) if the device's energy level is below certain threshold than resolution of location sensing parameters is reduced (iv) if multiple location sensing applications are running then the one with finer sensing parameters is used for location sensing and all other applications are forwarded this location information. EnTracked [KLGT09] and other approaches such as [BAPH09] achieves energy efficiency by using dynamic duty cycling based on usage of device's GPS or accelerometer.

Audio based Applications

SpeakerSense [LBBP+11] is a speaker recognition application. It uses heterogeneous multi-processor (HMM) architecture for mobile phones. Using this architecture continuous sensing of audio sampled from microphone is processed by a separate low power processor resulting in less frequent usage of high power processor on the phones. LittleRock [PLL11] employs a similar approach. It uses an extra micro-controller with different sensors such that the sampling and processing of sensor data is delegated to this micro-controller and the main processor or main micro-controller is not used for this purpose. SoundSense [LPL+09] is a speech and music classification system. As recognition of speech and music require computationally expensive functions and continuously executing such a system can drain the phone's battery quickly, SoundSense uses the entropy of the incoming audio frames to decide whether they contain meaningful energy after which they are further processed.

4.4 Gaps in Existing Work

In this section we discuss the gaps in the existing work mainly related to the context recognition frameworks and energy efficiency techniques and discuss the possibilities to bridge them in order to achieve our generic and energy efficient context recognition framework for smart phones. The identified gaps will help us to identify design requirements which should be fulfilled by our framework.

4.4.1 Context Recognition Applications

The examples of context and activity recognition applications using smart phones and wearable computers show the breadth of work that has been done in this field. The plethora of applications from different domains manifests the need as well as dependency on such applications by the user in the present as well as in the future. As miniaturization of the hardware platforms continues, the goal of seamless and distraction free user task support as envisioned by Mark Weiser becomes closer to the reality. The large number of existing and future context recognition applications also presents challenges to the research community. The main challenge lies in the fact that these applications are created independently by different developers in isolation which leads to their inefficient execution by resource constrained devices such as smart phones. Another challenge lies in the fact that context recognition applications usually follow similar design and to certain level similar functionalities which results in different implementations of essentially the same functionalities. As a result, the reuse of existing work is limited. These challenges can be addressed if the context recognition applications are developed using a uniform approach so that applications from different developers can be executed in an efficient way and the reuse of existing implementations can aid rapid prototyping.

4.4.2 Context Recognition Frameworks

The context recognition frameworks discussed in this chapter highlights need for system level support for the creation, execution and optimization of context recognition applications. These frameworks highlighted a number of requirements which should be fulfilled by a generic context recognition framework. The first requirement evident from the discussion of these frameworks is to have a uniform approach for the execution of these applications. As shown by [BKLA06] and [SDA99], the applications use a component based approach or a widgets based approach independent of their target contexts. Another requirement highlighted by these systems is the ability to extend the existing feature sets so that applications with new requirements can be incorporated. If a framework does not support incorporations of new applications then its usability will be very low. Another important feature is to allow applications to be configurable so that the same implementations can be used for different application requirements. However, the discussed frameworks do not provide a solution which possesses all these requirements and is suitable for smart phones. The ContexToolkit, CRN tool box or BeTelGeuse are designed for wearable computing

based applications and thus their design is inherently fine tuned for such applications. For example, in a wearable computing scenario using body worn accelerometers and audio sensors, the processing of the data would require management of different data streams over different communication interfaces. Similarly, the automatic reconfiguration supported by PCOM requires different devices to exchange contract information and communicate over changing technologies e.g. WiFi, Bluetooth etc. The smart phones are different from other wearable computing devices because all the sensing sources are embedded in the same hardware and managed solely by the local operating system. This alleviates management of remote communication and its associated overheads. Therefore, a framework for generic context recognition for smart phones can have a solution which has a simpler implementation and execution model for context recognition applications as compared to the wearable computing or distributed computing systems and applications.

The discussed frameworks also showed that use of graphical editors and tool chains for off-line analysis and optimizations of context information can increase developer's productivity and can also enable better user experience. Except for few, most of the existing frameworks do not offer such tool support. Therefore, a generic context recognition framework for smart phone should also be equipped with set of off line tools so that the rapid prototyping and optimization of existing solutions can be achieved.

4.4.3 Energy Efficiency Techniques

The examples of system based and application specific energy solutions show the impact of limited battery of smart phones on the design and execution of context recognition applications. These examples showed various strategies adopted for different requirements of the target context. The proposed solutions are useful and should be used in tandem when possible. However, we notice that as there is increasing number of context recognition applications targeting different context and to some extent these applications also provide energy efficiency mechanisms, executing such applications simultaneously may lead to faster battery depletion. Though there are frameworks and systems as discussed which provide energy efficient execution for multiple applications. However, the existing solutions do not tap into the fact that context recognition applications usually follow somewhat similar recognition logics and application structures. Thus, it is also possible to achieve energy savings when similar computations across different applications are done once and results are shared across the applications. Moreover, existing work also suggests that there are certain common techniques which are used by different applications and systems. These works also suggest that certain techniques are proven to be beneficial for certain context. These techniques include use of low cost operations instead of high cost operations when accuracy is not compromised, determination of context in series of steps, changing the context recognition accuracy based on user preferences, for example. Therefore, a comprehensive solution should also provide a generic support for such techniques such that they can be used by applications targeting different contexts.

Frameworks	Uniformity	Extensibility	Configurability	Energy Efficiency(*)
Context Toolkit	✓	✓	✓	X
CRN Toolbox	✓	✓	✓	X
PCOM	✓	✓	✓	X
BetelGeuse	✓	—	✓	X
ContextPhone	—	✓	✓	—
SeeMon	✓	—	—	X
EEMSS	✓	✓	✓	X
ACE	✓	✓	—	X
SymPhony	✓	—	✓	X
CenceMe	✓	—	✓	X
[ZKS10]	✓	X	✓	X
Desired Framework	✓	✓	✓	✓

Table 4.1: Classification of context recognition frameworks and energy efficient systems

We have summarized the discussed shortcomings of the existing work in a tabular form. In the table, we compare the discussed context recognition frameworks and systems which support energy efficient techniques against four design attributes identified from the survey of the existing literature. These attributes include (i) Uniformity(uniform design of applications and their execution), (ii) Extensibility (ability to add new context recognition features to the applications and the framework), (iii) Configurability (ability to change application settings) and (iv) Energy Efficiency (ability to support energy efficient execution of multiple applications executing simultaneously by removing redundant functionalities between them). The entries in the table are marked using three different symbols. The symbol (✓) denotes that the framework or the system possess that attribute. The symbol (X) denotes that the framework or the system does not possess that attribute and the symbol (—) denotes that the attribute is either not applicable to the framework or the system or either its applicability has not been discussed by the respective authors.

The resulting table is shown in Table 4.1 which shows the fulfilment of different context recognition frameworks and energy efficient context recognition systems against the four attributes. We can see that most of these systems and frameworks qualify for uniformity, extensibility and configurability. However, majority of these systems and frameworks are designed for distributed environments and wearable computing domains and hence are not directly applicable to context recognition for smart phones. Though, some of systems do provide means for achieving energy efficient execution of the applications but they do not exploit the redundant functionalities when multiple applications are executed simultaneously. Hence, the framework to provide generic and energy efficient context recognition for personal mobile devices should possess all the four attributes.

[0](*) Energy efficiency for executing multiple applications simultaneously by removing redundancy

The table does not show coverage of the context recognition applications because the applications are usually fine tuned for their target context which means that their design does not have to follow a certain standard as the applications are not part of any framework and thereby they do not require same execution environment. As a result they do not fit to our criteria of Uniformity. Similarly, these applications also do not fit the criteria of Extensibility and Configurability as their main goal is to recognize a specific context, therefore, extending them to incorporate new functionalities or configuring them for optimizations is usually not the main focus of their development.

4.5 Summary

In this chapter we have discussed a number of context recognition applications for smart phones and wearable computing domain to demonstrate the interest they have generated for the research community as well as for the end users. The popularity of such applications for different domains has led to the development of frameworks to support the implementation of those applications. For the efficient management of energy resources of smart phones, the existing solutions are usually targeted at the application level (customized solutions only usable for the application) or at the system level (solutions available for different applications using that system). We have identified that the solutions for the development of context recognition applications and their energy efficient executions is though supported by different frameworks and systems, there is however a need for a more integrated solution which can address the challenges we have identified for providing generic yet energy efficient context recognition for smart phones. Based on the gaps identified in the existing work, in the next chapter we define the architecture for our framework followed by its details in the subsequent chapters.

5 FRAMEWORK OVERVIEW

In this chapter we give an overview of our generic and energy efficient context recognition framework. We begin with discussing different design requirements that we have identified through the literature survey performed in the previous chapter. These design requirements are necessary for realizing our framework. Using these requirements as guidelines, we give an overview of the framework architecture and discuss the design rationales behind its building blocks. At a high level, the framework consists of a runtime system and off-line development tools. The runtime system consists of two subsystems namely the component system and the activation system whereas the off-line development tools comprise of graphical editors, code base for commonly used context recognition functions and code generation utilities.

5.1 Framework Design Requirements

Based on the current state of the art on context recognition using smart phones discussed in the previous chapter, we have identified four design requirements which must be fulfilled by our framework in order to support the generic and energy efficient context recognition. These four requirements are Uniformity, Extensibility, Configurability and Efficiency. In the following we discuss them in detail.

5.1.1 Uniformity

Uniformity means the ability to recognize different context features in a similar manner. This means that the context recognition applications must adhere to a particular design approach for their implementation. As a result, their execution platform can execute these applications systematically and acquire the recognized context in a consistent manner. For example, a navigation application might be interested in the speed of a user to switch between visual and audio modes where as a calendar application might be interested in the surroundings of a user to determine whether and how an upcoming appointment should be signalled to the user. As both of these applications are interested in different contexts, and if they are developed without adhering to a particular design approach, their execution would not be possible in a systematic way which would prevent benefiting from advantages which could be provided with their combined context information e.g. if the user is driving then the system should not signal him/her the upcoming scheduled appointment from the calendar. Thus, in order for our framework to be useful for broad range of applications and

scenarios, a generic and energy efficient framework for context recognition should support the development of applications and recognition of different context features in a uniform manner.

5.1.2 Extensibility

The modern day context recognition applications target different context features such as location based, movement based, activity based contexts etc. Since there can be numerous context features that may be needed by different applications, it is unrealistic to assume that a single developer is able to provide context recognition methods for all. As a consequence, our context recognition framework should be extensible and it should enable the joint use of context recognition methods that have been developed in isolation. Furthermore, to ease the development of new context recognition methods, the framework should enable the reuse of parts of the existing methods when appropriate. For example, we assume that a developer A implements a context recognition application which determines user's mode of transportation i.e. whether user is driving a car, travelling in a bus, etc. We also assume that another developer B implements a context recognition application which determines modality of user's physical movement such as running, walking, standing, etc. Since these two applications are developed by different developers in isolation, if developer A would want to extend his application to support recognition of physical modalities and developer B would want to extend his application with recognition of motorized transport, they both would have to implement the respective features themselves. On the contrary, if both developers create those applications using a uniform approach and a same framework, they can extend their applications by reusing the existing parts from one another and at the same time contribute to the set of already implemented functionalities.

5.1.3 Configurability

Configurability means the ability to change the settings of a context recognition application such that the same application can operate differently. The context recognition methods used by context recognition applications often propose a particular set of parameters to balance the trade-off between accuracy and resource utilization, for example. Yet, this optimization is usually scenario specific. In practice, different parametrizations may be optimal for different scenarios. Moreover, depending on the scenario there may be several alternative methods for recognizing the same context feature. As a simple example consider that the location of a user can be detected using different technologies such as GPS in outdoor scenarios or WiFi in indoor scenarios. Therefore, a location recognition application should be able to use different technologies when possible in different scenarios. In addition to using different recognition methods for different scenarios, the application should enable same recognition method to be configurable e.g. for indoor localization using WiFi, the detection of user movement is either done by sampling accelerometer continuously or with some intervals. Taking these functionalities into account, our framework should be

highly configurable such that the applications developed using it are applicable in a broad spectrum of scenarios.

5.1.4 Efficiency

Due to the use of personal mobile devices such as smart phones as the technical basis for framework, the resource usage for context recognition must not hinder the primary function of the devices such as calling, messaging, surfing the web etc. As a consequence, our context recognition framework must make efficient use of resources, in particular the battery consumption of the device. Towards this end, the recognition framework should not spend resources for recognizing irrelevant features. In addition to that, it should perform the recognition of the relevant features with minimal effort. Therefore, the framework should provide algorithms to enable efficient execution of the applications. In addition, the framework should also provide means for the developers to efficiently create new context recognition applications. Therefore, the framework should provide set of off-line development tools in this regard.

5.2 Framework Architecture

An overview of the architecture of our context recognition framework is illustrated in Figure 5.1. As shown in this figure, the framework consists of two main building blocks, namely the runtime system which is deployed on a smart phone and an associated set of development tools that is used off-line to create the recognition methods for context recognition applications. The task of the runtime system is to perform the actual recognition of the relevant features of the context and it enables applications to retrieve them and to be notified upon changes. The task of the development tools is to enable the development of new recognition methods. To simplify this, the development tools facilitate the reuse of recognition logic that has been developed previously. The development tools are also used to train the classifiers for the applications. In the training phase, the sensing components are used to gather traces. These traces are then used with different preprocessing functions to develop the classifiers. In the following, we describe both, the runtime system and the development tools in greater detail and we discuss how they interact in order to realize the requirements.

5.2.1 Runtime System

The main task of the runtime system is to perform the context recognition. To do this, the framework introduces two different systems namely the component system and the activation system. As shown in Figure 5.2, the component system forms the lower level of the runtime system. On top of the component system, the framework introduces an activation system which uses a state machine abstraction that is realized by a minimal

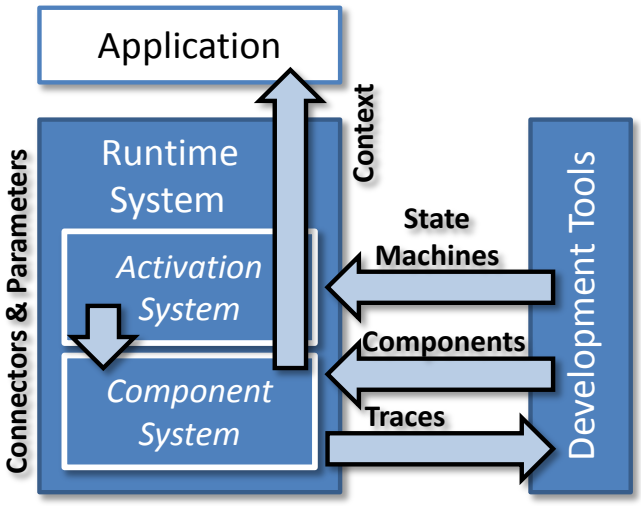

Figure 5.1: Architecture of generic context recognition framework

activation system engine. The main task of the component system is to perform the recognition of a set of context features in an efficient manner. To do this for arbitrary features, the component system abstracts form the specific recognition method by introducing three different abstractions, namely components, connectors and configurations. The main task of the activation system is to enable context dependent activation of context recognition applications such that at a particular time instance only relevant applications or only relevant context recognition methods are executed. In the following we provide rationales for using the component abstraction and the state machine abstraction in our framework and discuss how together, they can fulfil the design requirements.

5.2.2 Design Rationale for Component System

Users of ubiquitous computing systems are often involved in multiple tasks at a time which requires a simultaneous execution of several context recognition applications. As a simple example consider that when driving around, a user might want to use both, the travel time as well as the road monitoring application, while using the social networking application to update the status with the music playing on the radio. As a result, the applications have to continuously sample and process acceleration data for road monitoring and speed estimation as well as audio data for music classification and honk detection. Yet, since running multiple applications simultaneously increases the energy requirements additively and due to the fact that mobile devices are usually battery powered, this approach is inherently limited in scale.

When executing multiple context recognition applications simultaneously, additive increases in energy requirements can often be avoided. The reason for this is twofold. Firstly, existing applications are often using overlapping sets of sensors which are operating on low duty cycles. As a result, it is possible to reduce their combined energy consumption by

avoiding duplicate sampling. To do this, it is necessary to schedule sensors in such a way that their samples can be used by several applications. Secondly, the signal processing methods applied by different applications are often similar. Consequently, it is possible to reduce the energy requirements by avoiding duplicate computations. To do this, it is necessary to identify identical (intermediate) results and to replace all associated computations with a single one.

Clearly, from a theoretical point of view, it is possible to exploit the overlapping sets of sensors and duplicate computations by manually integrating a targeted set of applications. However, since not all applications are required at all times by all users, the manual integration of context recognition applications would require a significant development effort that increases exponentially with the number of used recognition applications. As a consequence, we argue that the exploitation of overlapping sets of sensors and duplicate computations must be automated.

As a first step towards such automation, it is necessary to detect duplicate sampling and processing code which requires an analysis of the applications that are supposed to be executed simultaneously. Intuitively, the complexity of such an analysis depends on the degree of structure applied to the application. If an arbitrarily structured application shall be analyzed, the overall problem is at least as complex as the detection of code clones [SWP+09],[GJS08],[MM01]. Due to the associated computational effort, such an analysis can only be done offline on a powerful device. Consequently, to support the online analysis on a personal mobile device, it is necessary to further structure the applications.

From the set of possible approaches, we chose a component based approach which requires application developers to structure their applications by composing a set of components. The reason for this is threefold. First, there already exist rapid prototyping toolkits such as [BKLA06] that validate the suitability of component abstractions to develop context recognition applications. Second, given a sufficiently large repository of standard components, a component-based approach can significantly speed up application development through reuse which at the same time ensures the effectiveness of duplicate removal. Last but not least, given a suitable connector model, the components that constitute an application can easily be manipulated at runtime.

To avoid unnecessary restrictions, we chose to apply the component structure only to the parts of the context recognition application that deal with the actual recognition. Other parts such as user interfaces, networking logic, etc. can be structured arbitrarily. Consequently, our approach introduces a clear separation between the recognition of context characteristics and their further use.

The use of component abstraction not only enables energy efficient execution of context recognition applications and fulfils design requirement on efficiency, it also provides uniformity of design for the applications. The uniformity of design results in re-usability and consequently fulfils requirement on extensibility. Moreover, as applications are created using components, configurability of applications at fine level of granularity is also possible.

Component Abstraction

The component abstraction used by our framework comprises of components, connectors and configurations. In the following, we briefly describe their structure and purpose:

- Components: In our framework, the components are the (reusable) building blocks of the recognition methods. Conceptually, a component consists of an implementation that exposes typed input and output ports that may be connected to the ports of other components. Thus, the ports enable components to interact with each other in a controlled manner. To do this, a component may post a value to an output port or it may read a value from an input port. In addition to ports, a component may expose parameters that can be used to adapt its internal behaviour. However, the parameters are not exposed to other components but, they can be accessed and manipulated by the component system itself. Developers may define different parametrizations for a particular usage of a component.

- Connectors: In order to be reusable, components are isolated from each other by means of ports. However, the recognition of a feature often requires the combination of multiple components in a specific way. Connectors express such combinations by determining how the input and output ports of a set of components shall be connected.

- Configuration: In general, it is possible to categorize the components according to their function into three different levels. As shown in Figure 5.2, at the sampling level a component may provide access to a physical or virtual data source that can be used to sample data. Components at the preprocessing level perform generic tasks such as noise-reduction or feature-extraction on the raw data provided by the sampling level. On top of the preprocessing level, classification components combine multiple features to recognize the target feature of the context. Except for the most basic methods, most feature recognition methods have to combine a number of components at all of the levels. We term set of components and connectors as a configuration. For example, an application for recognizing a noisy environment will require a configuration consisting of sampling component to capture audio from a microphone, a preprocessor component to extract the power level and a classifier component to decide whether the environment is rather noisy or calm. However, most of the components usually exhibit parameters that can be used to control their behaviour. Therefore, it is necessary to provide the parametrization support for each component. For the mentioned example, such a parametrization might define the sampling rate and frame size for the sampling component or the threshold for the power level used by classifier component.

In order to recognize multiple context features simultaneously, the component system is able to execute multiple configurations. In cases where the configurations share similar parts of the configuration, the component system tries to merge them intelligently to

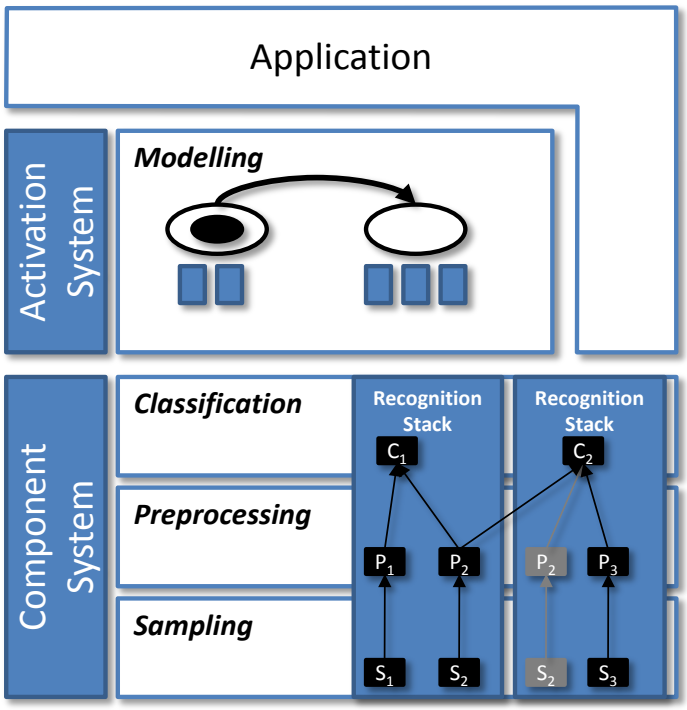

Figure 5.2: Runtime of generic context recognition framework

avoid duplicate computations. This ensures that the recognition is performed efficiently even in cases where the configurations have been developed independently. An example for this is shown in Figure 5.2. Since the depicted configurations share the sampling component S2 and the preprocessor P2, the component system only executes them once. In addition, it changes the connectors to use the output of the same preprocessor P2 in both configurations. Intuitively, the automated merging requires a runtime analysis of the configurations to detect the overlapping components in the two configurations.

5.2.3 Design Rationale for Activation System

Using the outlined component model and the associated merging mechanism, the component system is able to support the development and execution of arbitrary configurations in an efficient manner. Thereby, the component system executes the configured recognition continuously. However, depending on the state of the environment, an application may only be interested in a subset of the features at a time. In such cases, the continuous execution of all configurations would be inefficient. As for example, consider that a user uses two context recognition applications, one for determining his context when he is in his office and one for determining when he is driving. The user usually requires both applications five days a week but does not need them to be executed simultaneously. Therefore, if both of these applications are executed together when one of them is not needed, it will result in the wastage of resources.

To avoid this, the framework enables applications to specify the desired configurations in a context dependent manner. This is handled by the second system of our framework, the activation system. Using the activation system, an application specifies a state machine description of states to model requirements on the recognition to model ways of switching between the states where states are associated with different configurations. For the mentioned example, by using the activation system only one of the application i.e. either one for determining context when user is in office or the one for determining when the user is driving in executed at a particular time. Although this may seem complicated at first, it enables the application developer to apply fine-grained control over the recognition. If this fine-grained control is not needed, it is possible to specify a static set of requirements by introducing a single state without transitions.

State Machine Abstraction

The state machine abstractions have been used in existing context and activity recognition systems. Examples include [FCCRS12] [TJDS09]. [FCCRS12] uses state machine abstraction for the determination of suspicious human activities based on video surveillance. [TJDS09] uses finite state machines for detection of common household activities such as cooking, eating, brushing teeth etc. The system observes location of users by video cameras fitted on the ceilings and their actions with the help of a wearable sensors placed on users arms. These example system shows that the applicability of state machines in context and activity recognition is well established in the existing work. In the following, we briefly describe the state machine abstractions used by the activation system.

- State: State machines in the activation system consist of a set of states. Each state is associated to a particular requirement regarding the context recognition that shall be executed by the component system. To do this, each state may be associated with a set of configurations. One of the states is marked as initial state and this state is active when the state machine description is added. Conceptually, the states model different situations that require the detection of a distinct set of context features.

- Transition: In addition to states, the state machine may exhibit transitions between states. The transition can be associated with conditions over the context features that are currently detected. Intuitively, this reduces the set of features to the ones specified in the source states of the transition. A transition is taken if the source state is active and the specified context features are detected.

The runtime system of our framework introduces a simple state machine engine to execute the state machines. The main task of the state machine engine is to compute the active states and to trigger state changes when a transition is fired. To trigger these changes, the state machine engine relies on the component system to detect the context features specified in transitions. Once a state becomes active or inactive due to a transition, the state machine engine automatically stops the configurations defined by the inactive state and it starts the configurations of the active state. In order to start the recognition,

an application may inject a state machine description in the engine at runtime. Once the state machine is executed, an application may receive changes to the features that are detected by configurations that are executed by the component system. In addition to that, an application may also monitor the state transitions in the state machine engine. Thus, an application can use the states in the state machine to summarize a particular set of changes to context features.

Moreover, there exist some energy efficiency techniques in the literature. These techniques have been used for different contexts but no existing system provides their generic applicability. Using the state machine abstraction of our activation system it is possible to provide generic applicability of those techniques. Below we describe these techniques with respect to a localization example as described in [ZKS10] and later in the chapter on activation system we discuss how activation system provides their generic applicability.

- *Suppression:* In general, Suppression refers to the use of low energy consuming sensing sources first to determine whether the use of more energy consuming sources is beneficial. For a localization example, suppression refers to the use of low energy sensors to first determine the user movement and then use high energy sensors to perform localization. As an example consider a scenario where WiFi is used to determine user location. Instead of continuously sampling the WiFi sensor regardless the user has moved or not, the more energy efficient solution would be to first sample accelerometer sensor for detecting any movement. If a movement is detected only then the WiFi sensor is sampled and the location is determined. Therefore, Suppression can be generalized as the mechanism of using fewer resources to identify the needs for more resources.

- *Substitution:* In general, Substitution refers to the use of low energy consuming sources instead of high energy consuming sources when accuracy of recognition is not affected. For a localization example, Substitution refers to the use of low energy consuming location sensing sources instead of high energy consuming source, provided the accuracy of low energy consuming source is equal or better than the accuracy of high energy consuming source. As an example consider a scenario where a user moves from location A to location B. Along the route, the user has to go through indoor and outdoor locations. As GPS is usually more accurate in outdoors, therefore it is used as long as the user is outdoors. When the user moves to indoor, GSM or WiFi localization methods are used. In case the accuracy provided by GSM or WiFi in outdoors is same as GPS then GSM or WiFi is used instead of GPS. Therefore, Substitution can be generalized as the mechanism of using lesser energy consuming context recognition mechanisms when these mechanisms can provide equal or better accuracy of results compared to the mechanisms which require more energy.

- *Adaptation:* In general, Adaptation refers to adjustment of context recognition parameters based on factors such as current battery level. For a localization scenario, Adaptation would refer to the adjustment of location sampling parameters based on the current battery levels or other user settings. If the battery level is below certain

threshold then the location sensing time interval or location sensing distance interval is increased. As an example, consider a scenario where user is using a location based application which uses GPS for determining the location every one minute. As the battery level goes below a certain threshold the location sensing interval is increased to three minutes. As a result, the location based application is now notified about the location every three minute. Use of Adaptation may reduce the accuracy required by the application but at the same time saves energy for the longer operation of the phone. Therefore, Adaptation can be generalized as a mechanism of changing the settings of context recognition applications based on changing availability of other resources.

- *Piggybacking:* In general, Piggybacking refers to combining similar context recognition requests from different applications and perform recognition only once and share the results with all the applications instead of computing same context for all applications separately. For a localization example, Piggybacking refers to combining multiple location sensing requests in to a single request. If there are more than one location based applications asking for the current location, piggybacking uses the sensing parameters of one application which are finer than the sensing parameters of all other applications. It then reports the location information to all the applications based on the chosen parameters e.g. if there are two location based applications and one of them requires location information every 1 minutes while the other requires it every 2 minutes and both applications are started at different time instances then piggybacking will sample and provide location information every 1 minute to both applications (at the same time). Piggybacking reduces number of location sensing operation considerably if there are multiple location based applications started at different time intervals. Therefore, Piggybacking can be generalized as a mechanism to combine the provisioning of similar resources to different applications. In our framework, piggybacking is achieved by using configuration folding described in a chapter 8 of this thesis.

In order to provide generic applicability of these energy efficiency techniques beyond location sensing, a generic system must provide support for the following three functions:

- *Conditional execution:* In order to enable *Suppression*, the system must be able to recognize context in multiple steps (possibly structured in a hierarchical fashion) such that low energy consuming operations are performed first to decide whether high energy consuming operations are required. This can be done by enabling the conditional execution of different steps based on the context that has been recognized so far.

- *Adaptation support:* To enable *Substitution* and *Adaptation*, the system must provide adaptation support so that an application can modify the behavior of its context recognition logic by adjusting settings. This can be triggered either based on the cost for determining context (Substituion) or on the state of the device, e.g. its remaining battery power (Adaptation).

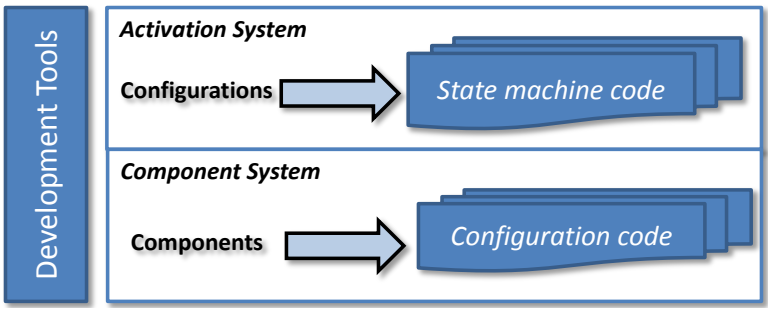

Figure 5.3: Tool support in the generic context recognition framework

- *Request multiplexing:* To enable *Piggybacking*, the system must be able to capture the requests from multiple applications and handle them jointly in order to avoid duplicate sampling and processing.

Using the state machine abstraction used in our activation system, it is possible to support these functions. Since a state machine introduces states and (conditional) transitions, they are a simple and flexible way to model the conditional execution. For example, developers can structure applications such that states represent low and high energy consuming operations and transitions represent the rules for switching between them. Similarly, to achieve adaptation support, developers can use states to represent alternate context recognition methods with different energy and accuracy cost to recognize same context or use them to represent different application settings relevant to particular battery levels. The transitions then provide switching between different recognition methods or different application settings. In addition, adaptation support can also be achieved by structuring applications as combination of multiple state machines instead of just one state machine. In this case the transitions provide switching between different state machines. Finally, to support request multiplexing, we rely on merging of configurations (configuration folding) supported by the component system as mentioned earlier. As the component system models individual context recognition actions as parametrizable component configurations that can be executed by a common runtime system. To enable piggybacking in a generic fashion, the component system applies configuration folding which dynamically analyses a set of simultaneously executed configurations in order to eliminate redundant functionalities in the configurations.

5.2.4 Development Tools

In addition to the runtime system, the framework also provides set of development tools to enable efficient development of context recognition applications. Due to the modularity of the runtime system, there are two tools each for the component system and the activation system and they target different aspects of efficient development of the context recognition applications. The abstraction for these tools is depicted in Figure 5.3. The tools for the component system consists of a graphical editor, a large code base for frequently

used functions in context recognition applications and code validation and code generation utilities. The graphical editor for component system is used for creating the configurations by allowing developers to simply drag and drop components on the graphical pane, parametrize them and connect them via connectors. Once a configuration is created, the code validation utility validates the configuration and the code generation utility generates its code so that the configuration can be directly used in the applications.

Similarly, the activation system is also equipped with a graphical editor and code validation and code generation utilities. The graphical editor for the activation system allows developers to drag and drop configurations (developed using the graphical editor of the component system) on the graphical pane and connect them to states. The editor allows the states to be connected using transitions. In this regards, the editor also allow creation of rules to be associated with the transitions. Once a state machine is created, the code validation and generation utilities are used for generating the code of the state machine. This code can then be directly used in the applications. A detailed description of these development tools is given in the respective chapters on the component system and the activation system.

5.2.5 Example Application

In order to assess the suitability of our architecture, we initially implemented the individual parts of the architecture and then a built a simple example application. The application has been written in Java [Jav] and targeted towards the Android [anda] platform. We implemented a simplified version of the runtime system which was later refined for additional recognition methods for different context features and is discussed in detail in the next chapters. The example application recognizes the location of the user using a smart phone that is running an instance of simplified version of the the framework. The application then visualizes the user's location in a moving map. Although, the application itself is not complicated, it uses many features of the framework.

Intuitively, there may be multiple alternative ways of retrieving the location information. In outdoor scenarios, the localization by means of GPS is a natural choice due to its wide availability and simplicity. However, in indoor scenarios GPS cannot be used. Thus, in order to provide location information in a seamless manner, it is necessary to use an alternative configuration. Due to its wide availability, a rather common approach is to use 802.11 technologies for indoor localization and there are several possible ways of doing this. For the example, we decided to use fingerprinting due to its simplicity. The bottom of Figure 5.4 shows the configurations resulting from the usage of these recognition methods.

The configuration for the outdoor localization, depicted on the left side of Figure 5.4, solely consists of a single component at the sampling layer. The reason for this is that the GPS hardware is already performing the necessary preprocessing and classification internally. Thus, the GPS component can simply retrieve the current coordinates from the GPS receiver which can then be made available to the application.

The configuration for the indoor localization, depicted on the right side of Figure 5.4, is more complicated. At the sampling layer, it consists of a WLAN component and a MAP

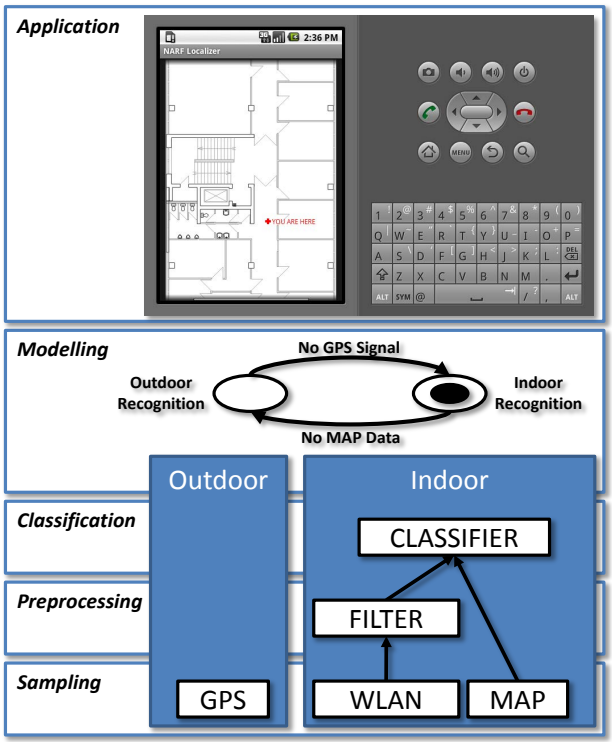

Figure 5.4: Example application using proposed framework

component. The WLAN component is responsible for generating fingerprints by scanning the available 802.11 networks to determine their signal strengths. In order to derive the current user location from the fingerprint, the MAP component provides access to a list of fingerprints with associated locations. To do this, the MAP component uses BASE [HBS03], a middleware for spontaneously networked devices which enables the component to retrieve the list from a local server. At the preprocessing layer, a filter component is responsible for improving the fingerprints by removing irrelevant access points. The resulting fingerprint is then used by the CLASSIFIER to determine the location using the list provided by the MAP.

Since a user can either be located indoors or outdoors at any point in time, running both configurations simultaneously results in unnecessary resource usage. To minimize the resource usage, the application specifies its requirements with a state machine that models the fact that the two configurations should not be executed simultaneously. This can be done by introducing a state machine description with one state for each configuration and appropriate transitions.

Intuitively, the transitions need to be refined with conditions that define when to switch between the states. As explained earlier, a key limitation is thereby that the conditions may only refer to context features that are detected in the source state of the transition. Otherwise, the state machine engine would not be able to detect the feature by means of the component system. Thus, for this example, we define the conditions in such a way that

they are triggered when the localization is no longer successful, i.e. if the GPS component does no longer receive coordinates or if the map component does no longer provide a list of fingerprints. Clearly, this may result in oscillations in cases where both, the indoor and the outdoor location, cannot be performed. However, in such cases the interleaved scanning is a desirable behaviour.

In order to use the recognized features, the application may access the location provided by the component system as well as the current state of the state machine engine. Our example application uses both. The state of the state machine engine is used to adapt the way how the application tries to retrieve a map. If the outdoor recognition is used, the application can simply retrieve a map through a web service such as Google maps. If the indoor recognition is used, the application uses BASE in order to download an indoor map from a server that is deployed locally in the current environment. The location provided by the component system is then used to mark the current user position in the map.

Requirements Coverage for Example Application

In the following, we discuss how the architecture of our framework addresses the requirements identified earlier in the chapter. To clarify the individual points, we revisit the example application described previously.

- Uniformity: As discussed previously, the framework architecture is independent from the context features that shall be recognized. It enables uniform access to the recognized features by means of supporting arbitrary access to all types of data that are generated within a configuration. The example application solely uses the information that is generated at the top most component of each stack. However, other applications might benefit from information that is generated by the intermediate components. As an example consider that it might be useful to visualize the list of available fingerprints to indicate the granularity of the localization. In addition, the framework enables applications to adapt the recognition using a state machine model. Although the primary use case for this restriction is to minimize the resource usage, exposing the current state of the state machine engine can provide valuable information. In the example application, the state of the state machine engine is used to switch between map providers. In general, it can be used for grouping the access to a set of context features.

- Extensibility: The framework architecture supports extensibility at several levels. At the component level, additional components can be implemented to provide access to physical data sources, such as a 802.11 interface, virtual data sources, such as a local map server, preprocessing algorithms and classifiers. Above the component level, developers can extend the framework by creating different configurations to recognize a context feature.

- Configurability: In order to adapt the recognition to a particular scenario, the framework architecture introduces state machines. The state machines describe which con-

figurations should be used at a time. By defining different parametrizations, it is also possible to use the same configuration with different trade-offs in a single application.

- Efficiency: To support resource efficient recognition, the runtime system of framework provides several manual tuning knobs and automatic mechanisms. Using state machines, it is possible to manually define the set of configurations that is relevant in a particular context. By determining the current state of the flow, the runtime system can then automatically starts and stops the configurations. Similarly, by enabling developers to provide different connectors and parametrizations, they can define different configurations to balance the trade-off between accuracy and resource usage. These can then be used as part of the state machine definition. Besides from reducing the resource utilization by optimizing the set of recognized features, the component system also minimizes the resource overhead resulting from the isolated development of configurations. To do this, the component system automatically merges the configurations, in cases where they share the same components. This enables the system to execute arbitrary combinations of configurations in an efficient manner.

5.3 Summary

This chapter has given an overview of the architecture of our generic and energy efficient context recognition framework. The chapter has defined four design requirements which our framework must fulfil. The chapter has discussed the design rationales behind the two systems of our framework, the component system and the activation system. The chapter has also discussed the importance of development tools for the framework. Towards the end, the chapter has used an example application created using a minimal version of our framework to show that the framework's architecture can fulfil all design requirements identified in the chapter.

6 COMPONENT SYSTEM

In this chapter we discuss in detail various aspects of the first system of our framework namely the component system. The component system provides basis for the development and execution of context recognition applications. We discuss the component model we used for the component system in detail. We present the runtime system of the component system and describe its building blocks. We also discuss the development tools associated with the component system. We describe their working and discuss the advantages they provide for the rapid application development. Lastly, we evaluate the component system by describing a number of applications that we implemented using it. Moreover, we also discuss how using these applications the design requirements for our framework are fulfilled.

6.1 Component Model

To structure the context recognition logic, our component system introduces a lightweight component model which introduces three abstractions namely, components, connectors and configurations. Components represent different operations at a developer-defined level of granularity. Connectors are used to represent both, the data- as well as the control flow between individual components. Configurations are used to define a particular composition of components that recognizes one or more context characteristics.

6.1.1 Components

Components represent atomic and reusable building blocks that constitute the context recognition logic. Example of context recognition logic could be a component which acquires audio data or a component which suppress noise from the audio data. To support application independent composition, each component may declare a number of strongly typed input and output ports. Input ports are used to access results from other components. Output ports are used to transfer computed results to other component. In case of an audio data sensing component, the output port of the component sends data to the input port of the noise suppressing component, for instance. As sending of data means passing of the resources such as buffers, threads, their management is performed externally by the component system. However, the component developer has to define the input and output ports for the components, the type of data they expect and actual logic of the component. In order to ensure that the actual logic is executed by the component

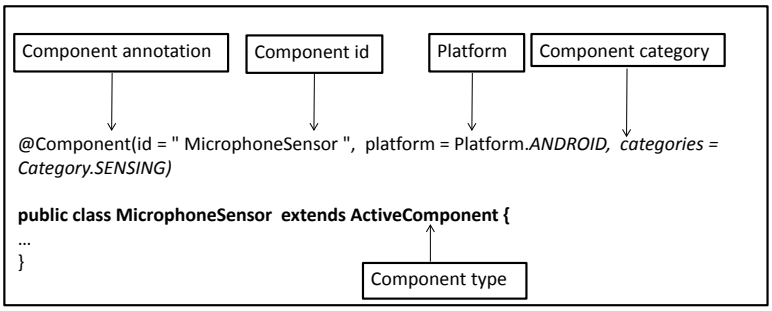

Figure 6.1: Component declaration for active component

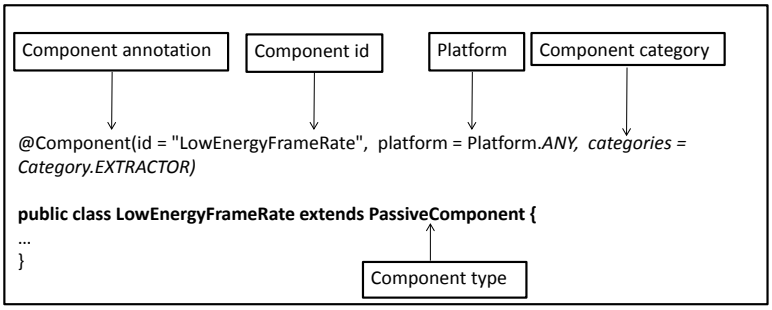

Figure 6.2: Component declaration for passive component

when it receives data on its port, the developer must provide code representing the logic in particular methods provided by the component system.

To simplify the reuse of the same component implementation across multiple applications, components can support a developer-defined set of typed parameters. For an audio data processing component at the sampling layer, the parameters might be used to express different sampling rates, sampling depths, frame sizes or duty cycles. Similarly, at the preprocessing layer, a filter component might use parameters to set low pass, high pass frequency values, for instance.

Active Component

The component system provides two types of components namely the active components and the passive components. An active component is a component which executes continuously unless stopped by the component system. An active component is not dependent on other components for its execution. Example of active component can be sensing components which does not require input from any other component and executes continuously or in a periodic manner and provides other components with some data. The component system manages the life cycle of the active components using onStart() and onStop() methods. The onStart() method is called by the component system to pass a thread of control to the component. The component is free to keep hold of the thread until the onStop() method is called using another thread. Once the onStop() method is called, the

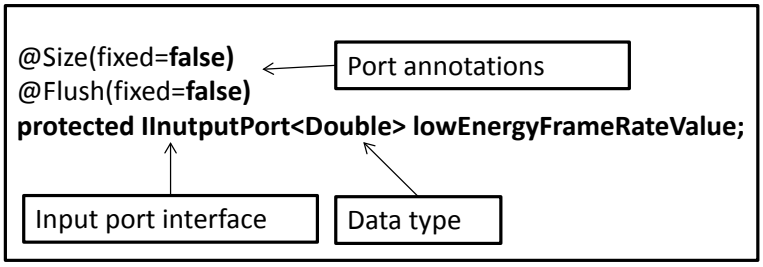

Figure 6.3: Input port of a component receiving low energy frame rate value

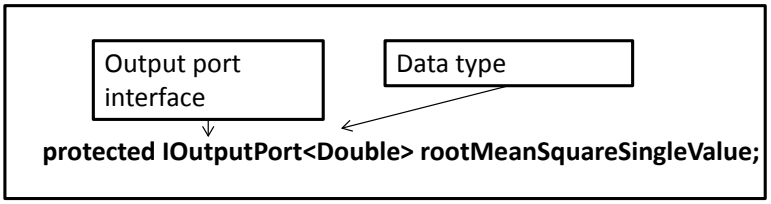

Figure 6.4: Output port of a component sending root mean square value

component should release the thread immediately (i.e. as fast as possible). An example of active component description is shown in Figure 6.1.

Passive Component

The passive component on the other hand, does not have its own thread of execution. Unlike the active components the execution of a passive component is dependent on the triggering from other passive or active components. An example for the passive component can be a windowing component which takes input from a sensing component and applies some windowing function to the data received from the sensing component. The execution of a passive component is managed by the component system on appropriate times. In order to determine appropriate times, the developer defines port sizes and flush values for each port. The port size indicates the total number of events that should be present on the input port of the component before the component can be executed whereas the flush size determines the number of events that should be removed from the component's input port after the component has been executed. The events can represent a data type or a collection of a data type. The component system manages the execution of passive components through onExecute() method. The onExecute() method is called whenever all input ports are filled with the required number of events. The required event number is specified via the port size variables and the calling interval is determined by the flush variable. Example of passive component description is shown in Figure 6.2 whereas the examples of description for input and output ports are shown in Figure 6.3 and Figure 6.4 respectively.

Component Category

We have created a large number of components using our component system. These components have been created for applications targeting different contexts. As mentioned earlier, a typical context recognition application consists of sensing, preprocessing and classification stages. Consequently, we group our components in different categories accordance with these stages. As sensing and classification stages are self explanatory and contain fewer components compared to sensing stage, we have used two categories to represent components accordingly. On the other hand, the preprocessing stage may have many components for different tasks. The components in the preprocessing stage may involve components for synchronization of data streams from different sensors, components for creating windows of fixed length of data coming from sensors, components for interpolating missing sensor values, components for extracting features from the sensor data etc., therefore, it is necessary to group the components for preprocessing stage in more finely defined categories. Therefore, we have used different component categories for our preprocessing components and the list is shown in the Table 6.1. The names used for the categories have been chosen to represent set of components which perform similar functions. The naming of categories is not fixed and can be changed. We chose these names because in our view they provide better classification. It is also noteworthy to mention that this list of categories is neither complete nor static. If a developer creates components whose functionalities cannot be represented by the existing categories, the developer can extend the list by creating new categories. Similarly, the classification of existing component to these categories in not static and the developer can change the categories of existing components as well.

Component Annotations

We use different annotations in a component definition. These annotations are used by the component system to manage the operations of the components. These annotations include @Component, @Flush, @Size, @Ignore and @Optional.

- @Component: This annotation describes the component information such as the identifier of the component, the platform on which the component can be execute and the category of the component. This annotation is used to define component structures at compile time. Example of component annotation can be seen in Figure 6.2.

- @Flush: This annotation is used to control the flush size of input ports. If an input port does not exhibit this annotation, the default values are applied. Example of flush annotation is shown in Figure 6.3.

- @Size: This annotation is used to control the size of input ports. If an input port does not exhibit this annotation, the default values are applied. Example of size annotation is shown in Figure 6.3.

Category Name	Category Description
SENSOR	These components belong to the sensing stage of context recognition e.g. audio sensing component collects audio data from microphone of the device.
CLASSIFIER	These components belong to the classification stage of context recognition and classifies data into different contexts e.g. speech recognition classifier determines if the audio data is speech or noise or an activity recognition classifier which determines if the accelerometer data represents some physical activity or not.
SYNCHRONIZER	These components perform synchronization e.g. a component which synchronizes data streams from different sensing component.
FRAMER	These components generate frames or reorganizes them e.g a windowing component which cuts data stream into frames.
INTERPOLATOR	These components interpolate frames e.g. a component which takes input from 16kHz audio sensing component and interpolate missing values due to the sampling rate of the audio sensing component.
TRANSFORM	These components perform different time and frequency domain related transformations.
EXTRACTOR	These components extract feature from the sensing data e.g. a bandwidth component determines the bandwidth of audio signal from its frequency domain data.
ACTUATOR	These components perform some from of actuation such as displaying a message or playing some sound e.g. a vibration component which puts the device in vibration when a certain event is detected.
CONVERTER	These components perform data type conversion e.g. a component which converts Short values to Double values.

Table 6.1: List of component categories

- @Ignore: This annotation determines whether the port should be ignored. This can be used to exclude ports from being picked up by the component systems analysis code. Typically, this is used to hide some fields from the reflection done by loaders or the visual editor.
- @Optional: This is an annotation for input ports that denotes whether the port is optional. Optional ports do not have to be connected and will not be considered when evaluating the execution rules of passive ports. If a port does not exhibit this annotation, the default values are applied.
- @Parameter: This annotation declares a parameter. It must be applied to all fields of a component that shall be picked up as a parameter by the component system.
- @TransformableParameter: This annotation indicates that the parameter can be evaluated for the use of a transformation component.

Component Example

To elaborate the discussion, we take the code of a component as an example and show how does a component in our component system looks like. The Figure 6.5, Figure 6.6 and Figure 6.7 show the code for an AudioSensor component. This component executes on Android based devices and samples the audio data from the device's microphone. The @Component annotation in Figure 6.5 shows that the component identifier is "AudioSensor", the platform type is ANDROID which means that this component can only be executed on Android based platforms which provide audio sensor and the category of this component is SENSOR which means that the component is used for sensing audio data. The class AudioSensor extends from AbstractPeriodicSesor class which extends ActiveComponent class which means that the AudioSensor component is an active component. This component has no input ports but only one output port called audioFrame. The data type of the output port is short[] which means that the output of the component is array of short values. Figure 6.5 also shows different parameters marked using @Parameter annotations. The parameter framesize has a default value of 1024 which means that if the developer does not provide any other value for the framesize attribute, the default value of 1024 is used. Similarly, the component uses other parameters such as frameNumber, samplingRate, samplingDepth, audioSource, stereo etc. The component overrides a onInit() method of the AbstractPeriodicSensor class to perform checks before performing the actual sensing. The code snippet of onInit() is shown in Figure 6.6. This component overrides the onMeasure() method of the AbstractPeriodicSensor class to perform the actual audio sensing. As the component system manages the execution of active component using onStart() and onStop() methods which belongs to ActiveComponent class, the AbstractPeriodicSensor class extends this class. On onStart() call from the component system, the onMeasure() method of the AbstractPeriod class is called and the component begins sampling of audio data. Similarly, when onStop() method is called by the component system, the audio sampling is stopped.

```java
@Component(id = "AudioSensor", platform = Platform.ANDROID, categories = Category.SENSOR)
public class AudioSensor extends AbstractPeriodicSensor {
    /**
     * The output port to transmit raw audio data.
     */
    protected IOutputPort<short[]> audioFrame;
    /**
     * The size of the frames generated by the sensor.
     */
    @Parameter("1024")
    protected int frameSize;
    /**
     * The number of consecutive frames to capture in one sampling cycle. If
     * there is no sleep time between cycles, this value is irrelevant.
     */
    @Parameter("1")
    protected int frameNumber;
    /**
     * The sampling rate in Hertz. Android devices are only guaranteed to
     * support 44100 and 8000.
     */
    @Parameter("8000")
    protected int samplingRate;
    /**fff
     * The sampling depth in bits per sample. Currently only 8 and 16 bits are
     * supported.
     */
    @Parameter("16")
    protected int samplingDepth;
    /**
     * The audio source for recording audio. This parameter reflects the
     * corresponding values in MediaRecorder.AudioSource. Currently, the
     * possible values are 0 for default, 1 for microphone, 2 for voice uplink,
     * 3 for voice downlink, 4 for voice call, 5 for camcorder and 6 for voice
     * recognition.
     */
    @Parameter("1")
    protected int audioSource;
    /**
     * A flag to indicate whether the recording should be stereo.
     */
    @Parameter("false")
    protected boolean stereo;
```

Figure 6.5: Code example of audio sensor component 1/3

```
 * The audio recorder used to perform the recording.
 */
private AudioRecord recorder;

/**
 * A global lock to synchronize different sensors.
 */
public static final ReentrantLock lock = new ReentrantLock(true);

/**
 * Performs parameter checks and configures the recorder.
 */
@Override
public void onInit() {
    super.onInit();
    if (frameSize <= 0) {
        throw new IllegalArgumentException(
                "Frame size cannot be smaller than 0.");
    }
    int channel = stereo ? AudioFormat.CHANNEL_CONFIGURATION_STEREO
            : AudioFormat.CHANNEL_CONFIGURATION_MONO;
    int encoding;
    switch (samplingDepth) {
    case 16:
        encoding = AudioFormat.ENCODING_PCM_16BIT;
        break;
    case 8:
        encoding = AudioFormat.ENCODING_PCM_8BIT;
        break;
    default:
        throw new IllegalArgumentException(
                "Sampling depth must be 8 or 16 bits.");
    }
    int length = AudioRecord.getMinBufferSize(samplingRate, channel,
            encoding) * 2;
    length = Math.max(length, frameSize * 2);
    recorder = new AudioRecord(audioSource, samplingRate, channel,
            encoding, length);
}
```

Figure 6.6: Code example of audio sensor component 2/3

```java
/**
 * Records the set of frames. If sleep time is 0, this method will keep on
 * recording.
 */
@Override
protected void onMeasure() {
    try {
        int frames = frameNumber;
        while (frames > 0 || sleepTime == 0) {
            Event<short[]> result = audioFrame.getEvent();
            if (result.data == null) {
                result.data = new short[frameSize];
            }
            // if we do not have the lock, we need to get
            // it and enable the recording, since it is off
            if (!lock.isHeldByCurrentThread()) {
                lock.lock();
                recorder.startRecording();
            }
            int dataLeft = frameSize;
            while (dataLeft > 0) {
                int read = recorder.read(result.data, frameSize - dataLeft,
                        dataLeft);
                if (read < 0) {
                    throw new RuntimeException("Recording failed (" + read
                            + ").");
                } else {
                    dataLeft -= read;
                }
            }
            // if someone else wants to record as well,
            // we need to let him record
            if (lock.hasQueuedThreads()) {
                recorder.stop();
                lock.unlock();
            }
            audioFrame.sendEvent(result);
            if (isStopped()) {
                return;
            }
            frames -= 1; // next frame
        }
    }
```

Figure 6.7: Code example of audio sensor component 3/3

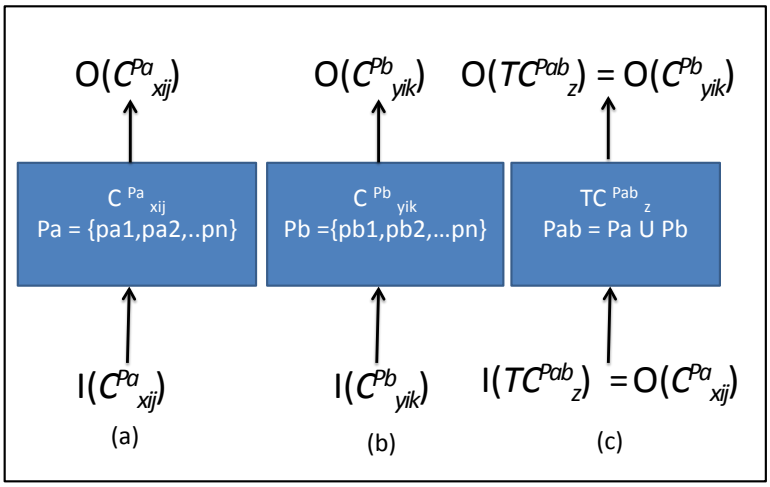

Figure 6.8: Component in configuration (a) j (b) k (c) Transformation component

6.1.2 Transformation Component

In order to support the energy efficiency using our novel technique configuration folding, we introduce a special type of component namely the transformation component. The transformation component is a component which can transform the output of a component under a particular parametrization to the output of the same component under another parametrization. In the following we discuss the definition and provisioning of transformation component. More details about their applicability is discussed in chapter on configuration folding.

Formalizing Transformation Components

Formally, we can define the idea of transformation components using a number of definitions as follows:

$C_{xij}^{P_a}$: *Component with id x at level i of configuration j*

P_a : *Set of parameters* in component $C_{xij}^{P_a}$

$I(C_{xij}^{P_a})$: *Input of component* $C_{xij}^{P_a}$

$O(C_{xij}^{P_a})$: *Output of component* $C_{xij}^{P_a}$

$ODT(C_{xij}^{P_a})$: *Output data type of component* $C_{xij}^{P_a}$

$IDT(C_{xij}^{P_a})$: *Input data type of component* $C_{xij}^{P_a}$

If $C_{xij}^{P_a}$ and $C_{yik}^{P_b}$ are two component in configurations j and k respectively such that x equals y but P_a does not equals P_b then transformation component for $C_{xij}^{P_a}$ is defined as following :

$TC_z^{P_{ab}}$: *Transformation component with id z*

P_{ab} : *Set of of parameters where* $P_{ab} = P_a \cup P_b$

$I(TC_z^{P_{ab}})$: *Input of Transformation Component*

$O(TC_z^{P_{ab}})$: *Output of Transformation Component*

$I(TC_z^{P_{ab}}) = O(C_{xij}^{P_a})$

$O(TC_z^{P_{ab}}) = O(C_{yik}^{P_b})$

$ODT(TC_z^{P_{ab}}) = ODT(C_{xij}^{P_a})$

$IDT(TC_z^{P_{ab}}) = ODT(C_{xij}^{P_a})$

As depicted in Figure 6.8 and also shown above, the output produced by the transformation component is same as the output of the component (in this case $C_{yik}^{P_b}$) it replaces. We can also see that the parameters of a transformation component consist of two sets of parameters corresponding to the parameters of both components. This knowledge of parameter values from both components is necessary for the computations performed by the transformation components. To elaborate the use of transformation component, consider an audio sensing component in a configuration samples data every x seconds and another audio sensing component in another configuration which samples data every 2x seconds. The transformation component for these two components will take in the output of the audio component which samples data every x seconds as its input and outputs every second value corresponding to the output of the component which samples data every 2x seconds. In order for the transformation component to work correctly, it needs to understand what sampling means and what should be the relation between the sampling of the two components in order for it to work correctly. Therefore, understanding of parameters and relations between their values is important for the execution of the transformation components. Further details on parametrization and relations between their values are given in the chapter on configuration folding.

Providing Transformation Components

The provisioning of transformation components is an additional development effort which includes selection of parameters, identification of relations between their possible values and

the implementation of the transformation components. There are two possible extremes regarding the responsibility of providing the transformation components i.e. either the (1) component system provides the transformation components or the (2) component developer provides them. Both extremes have their advantages and disadvantages. If the system provides the transformation components, then all the efforts lie at the system developer's end. In this case the component system provides the transformation components for the set of parameters known at the design and implementation time. The advantage of this extreme is that the component developer is relieved of any additional efforts where as the disadvantage is that this is a closed approach which means addition of new parameters and transformation components would mean extending or re-implementing the system. On the other hand, if the component developer provides the transformation components, then there is a flexibility to have new parameters and transformation components as per developer's need. However the disadvantage in this case is that it incurs additional development efforts.

Thus, instead of following one of these two extremes, we target a solution in the middle. Thereby, the component system provides the transformation components for the most commonly used parameters and at the same time allows developers to add the transformation components for the parameters they identify. This way, many typical applications can be supported without any additional development efforts on the components developers while still being flexible enough to deal with new parameter types.

6.1.3 Connectors

The typed input and output ports of different components are connected to each other using connectors. In order to minimize the overhead of the component abstraction, connectors are implemented using an observer pattern [GHJV94] in which the output ports are acting as subjects whereas the input ports are acting as observers. This enables 1:n relationships between components which are required to avoid duplicate computations. To avoid strong coupling between components, observers do not register themselves at their subject but the component system takes care of managing all required connections.

To avoid thread switching, a call to an observer not only forwards data, but it also passes the thread of control to the associated observer. Similarly, in order to avoid data copying, the subject passes its internal buffer directly to the observer. Consequently, observers may not write the data that is passed on from the subject. Clearly, from a software engineering perspective, both design decisions are dangerous as they require components to perform computations in a timely manner without manipulating their input data. However, given the associated resource requirements of frequent thread switches and copying between components, we opted for this less defensive scheme in order to facilitate more fine-grained components. Since components may be (indirectly) connected to several sampling components, components must be able to determine associated data received at different input ports. To do this, the data that is transmitted between components is encapsulated in an event structure. When required number of events are received at the non optional input ports of a component, the component logic is only executed then.

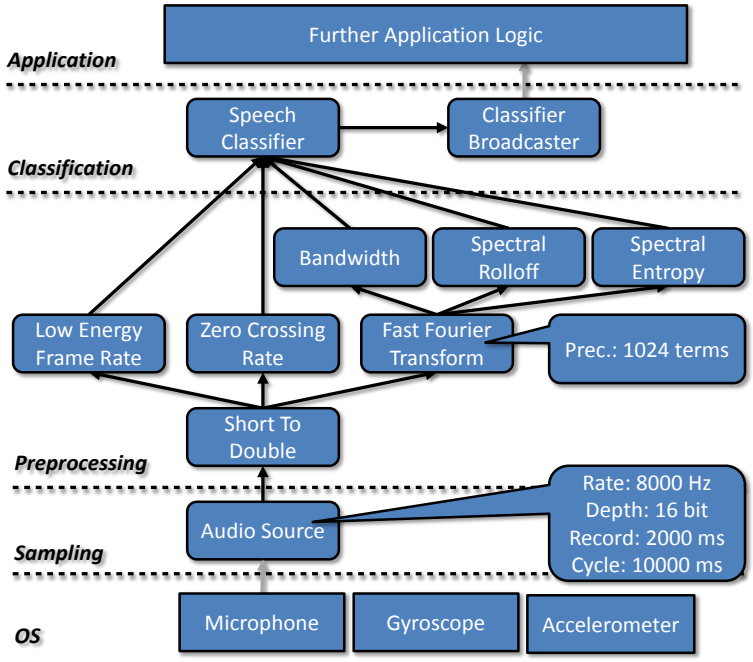

Figure 6.9: Example of speech recognition configuration

6.1.4 Configuration

Although components are equipped with typed input and output ports, the component system does not perform automatic type matching in order to connect components. Instead, each context recognition application must explicitly list all required components together with their connectors in a so-called configuration. In other words, the configuration represents set of components, their parameters and connectors between components such that they together constitute the context recognition logic. The developer of context recognition application creates the configurations and when instantiated, these configurations perform the recognition and report the result to the applications. While this approach slightly increases the development effort, it also increases the potential reuse of components that can be applied on data coming from different sources. As an example for such a component consider a Fast Fourier Transform which converts a signal from its time domain into the frequency domain. Clearly, such a component can be applied to various types of signals such as acceleration measurements or audio signals. Thus, by explicitly modelling the connectors of components as part of a configuration, it is possible to reuse such a component in different application contexts. An example of configuration for speech recognition is shown in Figure 6.9. This configuration consists of a number of components which are connected through connectors between the input and output ports of the components. The configuration shows AudioSource component at the sampling level. This component gathers audio data through system level calls. This component is parametrizable and its parameters are set with different values such as Rate is 8000 Hz, Depth is 16 bit, Record is 2000 ms etc.

These parameters can be changed by the developer of the configuration depending upon the context recognition requirements e.g the Rate can be set to 44100 Hz if better accuracy is required and battery consumption is not a concern. The audio data gathered by AudioSource component is passed to the ShortToDouble component in the preprocessing level. This component converts the data type of the audio data from short to double values and then pass the converted data to other time and frequency domain components. The time domain components in this configuration include LowEnergyFrameRate and ZeroCrossingRate component where as the frequency domain components include FastFourierTransform, Bandwitdth, SpectralRolloff and SpectralEntropy components. The outputs of these time and frequency domain components are forwarded to the SpeechClassifier component in the classification level. The SpeechClassifier component determines whether the audio data that is processed is whether a speech sample or not. The creation of configurations is supported by the development tools of the component system which alleviate the developers from creating the configurations manually. Detailed discussion on the creation of configuration is given in section on development tools.

Configuration Model

The configuration model uses component and connector models. The component model for defining components used in a configuration consists of component configuration, parameter configuration and port configuration. The UML (Unified Modelling Language) diagram of configuration is shown in Figure 6.10. The component configuration represents a particular component. It consists of name of the component, class name of the component, a flag to show whether the components has a delaying property for its transformation components (discussed in detail in chapter on configuration folding), a flag to indicate if the component has a transformation component and a string to represent the class name of the transformation component. The parameter configuration is used to represent parameters associated with the component and it consists of parameter's name, parameter's value, parameter type to indicate if the parameter belongs to a known type, the relation indicated possible relation values for the parameters if transformation component is to be used and parameter priority to show the priority for the parameters to evaluate the relationship according to the possible values. The port configuration consist of port's name, its type (input or output port) and the data type of the port. The connector model is represented by the link configuration and consists of the name of the source component, target component, name of the port of the source component and name of the port of the target component.

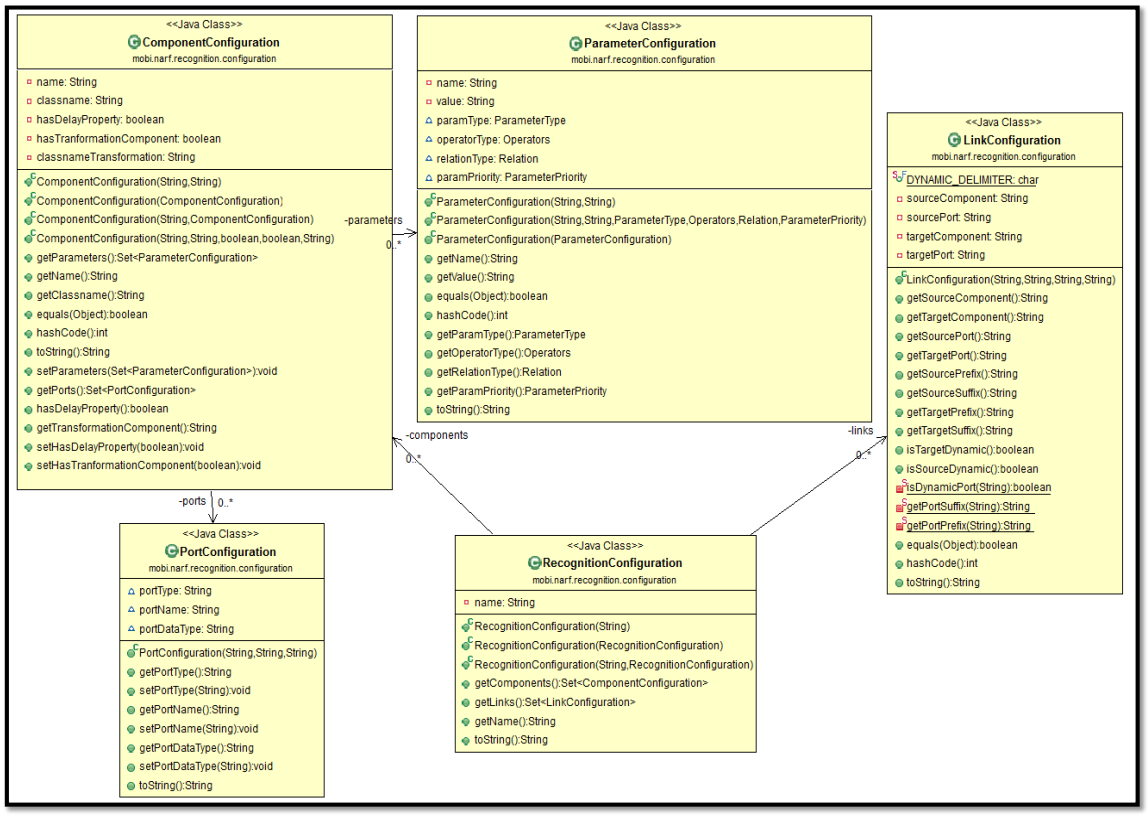

Figure 6.10: UML diagram of configuration model

6.2 Component System Execution

The fundamental building blocks of the component system are depicted in Figure 6.11. Each context recognition application consists of two parts, namely the part containing the recognition logic and the part containing the remaining application logic. The part that contains the recognition logic consists of a set of components that is composed according to a configuration. The part that contains the remaining application logic can be structured arbitrarily. Upon start up, a context recognition application passes the required configuration to the component manager which then takes care of executing the recognition logic in an energy-efficient manner. Upon shut down, the context recognition application removes its configuration from the component manager which will eventually release the components that are no longer required. As long as the recognition logic is running, the recognized context is signalled.

The runtime system of the component system comprises of different implementation modules namely the system module, component module, configuration module, template and platform module. The system module provides the runtime system support for the component system. It consists of a number of classes responsible for performing different functionalities required for the runtime system. These functionalities include loading of

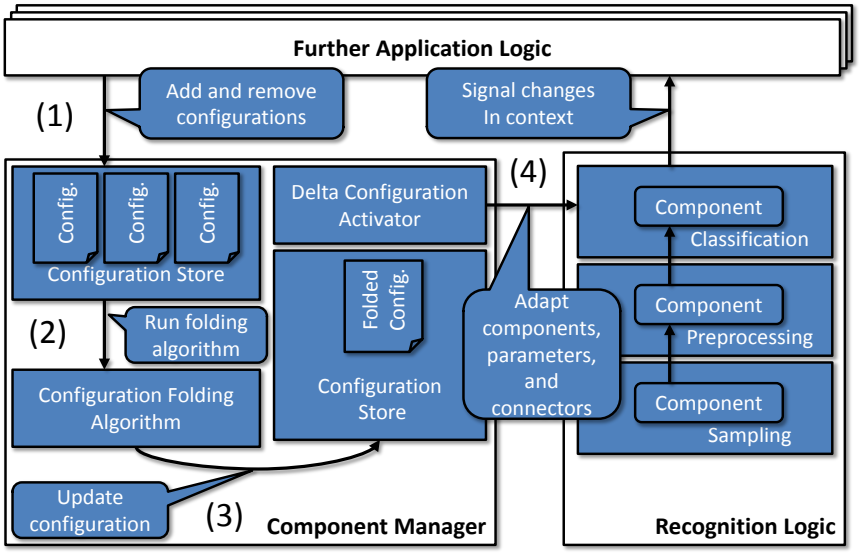

Figure 6.11: Execution of component system

components and parameters in the runtime system, validation of components and configurations, managing the registration and de-registration of components and configuration etc.

The system module class which manages the runtime system for the component system is called component manager. Specifically, it controls the execution of the componentized context recognition logic of all running applications. To manipulate the components executed at any point in time, the component manager provides an application programming interface (API) that enables developers to add and remove configurations at runtime. When a new configuration is added (Figure 6.11 (1)), the component manager first stores the configuration internally. Thereafter, it initiates a reconfiguration of the running recognition logic that reflects the modified set of required configuration. To reduce the energy requirements, however, the component manager does not directly start the components contained in the configuration. Instead, it uses the set of stored configurations as an input into our configuration folding algorithm (Figure 6.11 (2)) - which we describe in detail in the chapter on configuration folding. Using the set of configurations, the configuration folding algorithm computes a single, folded configuration that produces all results required by the application without performing duplicate sampling or computing duplicate intermediate results.

Once the folded configuration has been computed using the folding algorithm, the component manager forwards the configuration to the delta configuration activator (Figure 6.11 (3)). By comparing the currently running configuration with the folded configuration, the activator determines the set of operations that must be applied to the running configuration in order to transform it into the folded target configuration. Once the set of operations has been computed, the activator executes them. This may encompass the instantiation, parametrization and starting of new components, the stopping and destruction of com-

ponents that are no longer required, the stopping, re-parametrization and re-starting of modified components as well as the creation of new and the removal of existing connectors. When executing the different operations (Figure 6.11 (4)), the delta activator takes care of ensuring that their ordering adheres to the guarantees provided by the component life cycle. To do this, the activator first executes the stop calls, then it creates the new components and sets or changes the parameters, then it modifies the set of connectors and finally, it starts all components. To mention, the current implementation of delta activator stops the currently executed folded configuration and starts the newly folded one.

Since configuration folding cannot be reversed easily, the component manager performs a similar procedure once a configuration shall be removed. First, it removes the configuration from the internal storage (Figure 6.11 (1)), then it performs configuration folding (Figure 6.11 (2), (3)) and finally, it computes the required operations and executes them (Figure 6.11 (4)). Although, we could avoid this procedure in some cases by caching several previously folded configurations, we decided to use this simple procedure instead. The reason for this is that on the one hand, caching is only useful in cases where the removals are made in the inverse order of the additions. On the other hand, given that the number of active configurations is small, the overhead of repeated folding is marginal.

The component module of the runtime system of the component system provides client side code required to implement components. This includes abstraction for components and their parameters along with the annotations which provide meta information about the component. These annotations are used by the graphical editor to customize the components by altering parameter values. The configuration module provides client side code for creating configurations. As described earlier that a configuration consists of set of components and interconnection between them in some logical way, this module realize configurations consisting of component configuration, links configuration, parameter configurations and port configurations. Together, these sub configurations constitute a context recognition configuration which is used by the runtime system.

The template module provides system side code for managing components. Before instantiating the components and the links between them, the runtime system validates their syntactical correctness by conforming with the templates defined in this modules. The runtime systems then uses Java reflection utilities to instantiate the components and configurations. The platform module provides platform specific integration code for the component system. The component system can be executed on Andoird as well as J2SE platforms. However, depending upon the platform, certain functionalities cannot be executed on both. For instance, Android provides a set of APIs to call different physical sensors such as microphone or accelerometer.

6.3 Development Tools

The component system is equipped with a set of off-line development tools to facilitate the development of configurations to be used in the applications. These tools include a graphical editor for creating the configurations and a component tool kit consisting of a large number of components which developers can re-use to create new configurations.

6.3.1 Graphical Editor

The graphical editor for the component system enables rapid prototyping of configurations so that they can be developed quickly and used in the applications without having developers to write the code themselves. The graphical editor is implemented as an Eclipse [Ecla] plug-in using Eclipse modelling framework [Eclb] and Eclipse graphical editing framework [Eclc]. The graphical editor for the component system provides following functionalities.

Graphical Editing

The graphical editor enables developers to draw configurations for their application. As a configuration consists of set of components and connectors between them, this essentially means that the developers can create configurations on the graphical editing pane using components and connect them using connectors. In addition, the developers can also parametrize the components as per the requirements of their applications. In order to create the configurations, the developers can drag and drop the components on the graphical pane, parametrize them and connect them in a particular order. Figure 6.12 shows an example configuration. This configuration consists of 6 components connected in a particular order. The order of connection is dependent on the components and the context recognition logic which developer wants to achieve. In this figure, we can see that some of the components have their parameters exposed e.g. AudioSensor component has seven parameters namely component.stero, component.sleepTime, component.samplingRate, component.samplingDepth, component.frameSize, component.frameNumber and component.audioSource. The component.stero is a parameter which indicates whether the recording should be stero, component.sleepTime is the time between two measurements in milliseconds, component.samplingRate is the sampling rate in Herz, component.samplingDepth is the sampling depth in bits per sample, component.frameSize is the size of frames generated by the sensor, component.frameNumer is the number of consecutive frames to capture in one sampling cycle and component.audioSource indicates the corresponding values in MediaRecorder.AudioSource of Android where value 1 is for Android. The developer can set different values for these parameters to enable different functionality of the component.

Access to Component Toolkit

In order to create the configurations, the developers need access to the set of existing components. This could be done in two ways. One way could be that the developers

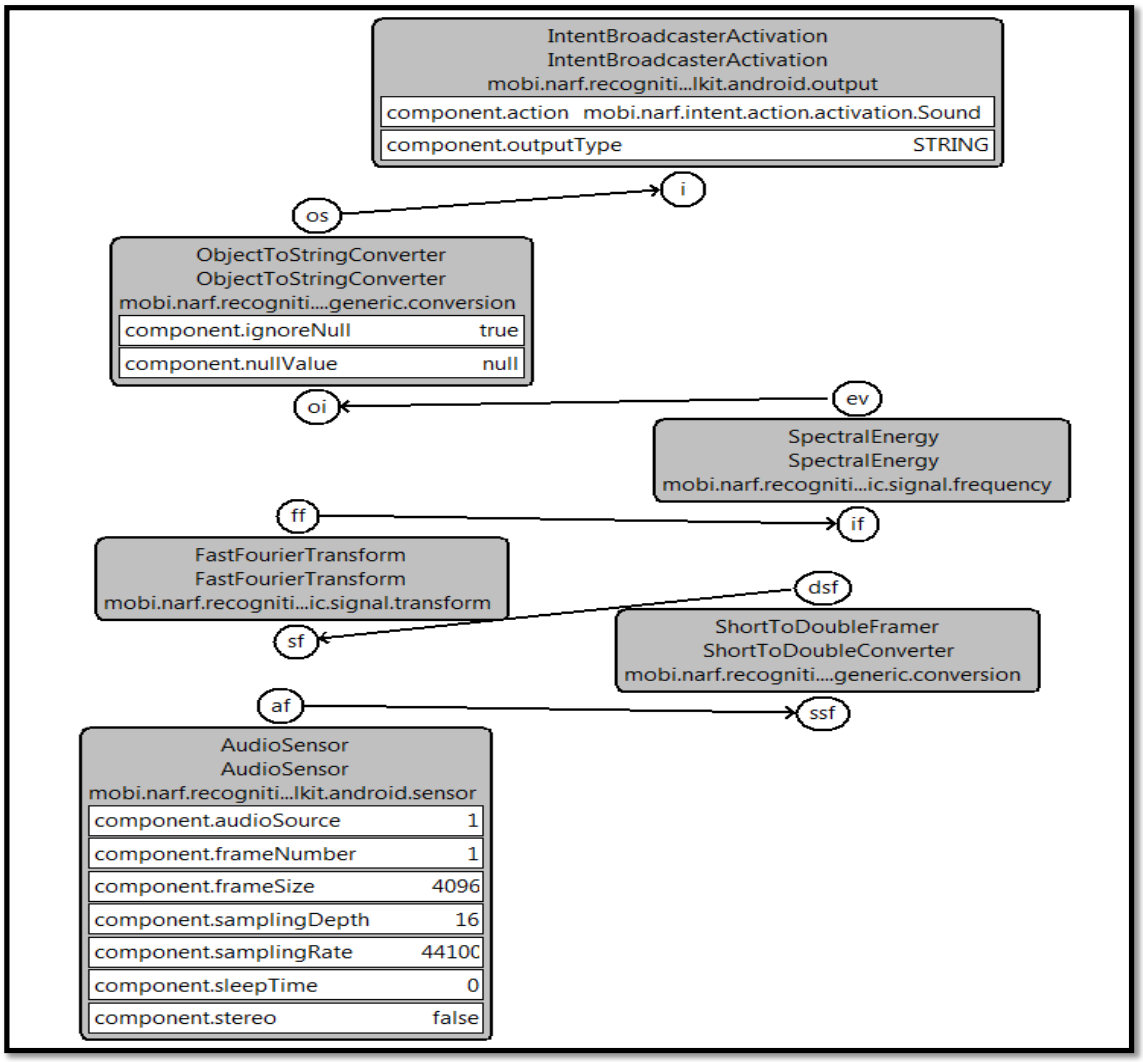

Figure 6.12: Graphical editor snapshot with example configuration

must check the class name and the name of the parameters manually for the components which are required for the configuration and create the configuration manually whereas the second way is that the developer can select the component from a list and simply click, drag and drop the component on the graphical editing pane. Clearly, the first approach is troublesome as it requires developers to manually check the implementation of each component whereas the second approach is more user friendly and productive. Therefore, our component system uses the second approach to enable access to the existing set of components. Figure 6.13 shows the snapshot of the list of components used by the graphical editor. We can see that the list consist of components is grouped under the categories defined earlier in this chapter. This way, developers can easily lookup the component categories, expand the one which is interested for them and use the component under that category in the configuration.

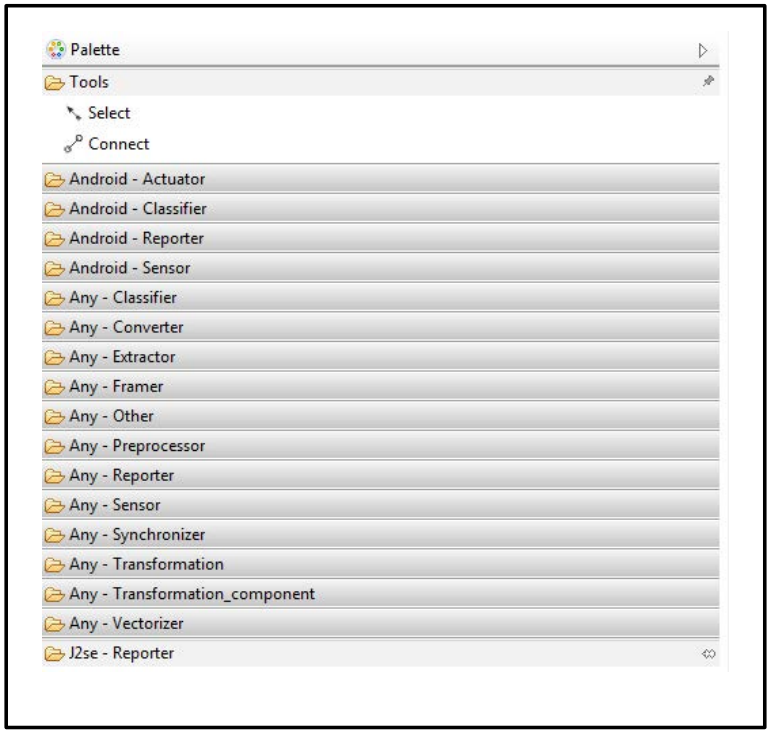

Figure 6.13: Component list in the graphical editor

Code Generation for Configurations

The configurations can consist of large number of components with different parametrization for the component. As a result, the code for the configuration can grow very large. Though, the code of a configuration involves similar syntax for each component and its connector, manual writing of code is troublesome and also less productive. Specially, during the training phase of the application when a developer creates different configurations to test their accuracies, writing the code manually is not an optimal approach. Conse-

```
public RecognitionConfiguration createConfiguration(String configurationName) {
    RecognitionConfiguration configuration = new RecognitionConfiguration(configurationName);
    // create component configurations
    ComponentConfiguration component0 = new ComponentConfiguration("FastFourierTransform",
        "mobi.narf.recognition.toolkit.generic.signal.transform.FastFourierTransform", false, false,"null");
    PortConfiguration component0Port0 = new PortConfiguration("fourierFrame", "OUTPUT","doubleArray");
    component0.getPorts().add(component0Port0);
    PortConfiguration component0Port1 = new PortConfiguration("signalFrame", "INPUT","doubleArray");
    component0.getPorts().add(component0Port1);
    configuration.getComponents().add(component0);
    ComponentConfiguration component1 = new ComponentConfiguration("ShortToDoubleFramer",
        "mobi.narf.recognition.toolkit.generic.conversion.ShortToDoubleConverter", false, false,"null");
    PortConfiguration component1Port0 = new PortConfiguration("doubleSignalFrame", "OUTPUT","doubleArray");
    component1.getPorts().add(component1Port0);
    PortConfiguration component1Port1 = new PortConfiguration("shortSignalFrame", "INPUT","shortArray");
    component1.getPorts().add(component1Port1);
    configuration.getComponents().add(component1);
    ComponentConfiguration component2 = new ComponentConfiguration("SpectralEnergy",
        "mobi.narf.recognition.toolkit.generic.signal.frequency.SpectralEnergy", false, false,"null");
    PortConfiguration component2Port0 = new PortConfiguration("energyValue", "OUTPUT","Double");
    component2.getPorts().add(component2Port0);
    PortConfiguration component2Port1 = new PortConfiguration("inputFrame", "INPUT","doubleArray");
    component2.getPorts().add(component2Port1);
    configuration.getComponents().add(component2);
    ComponentConfiguration component3 = new ComponentConfiguration("IntentBroadcasterActivation",
        "mobi.narf.recognition.toolkit.android.output.IntentBroadcasterActivation", false, false,"null");
    ParameterConfiguration component3Parameter0 = new ParameterConfiguration("component.action",
        "mobi.narf.intent.action.activation.Sound",ParameterType.OTHER,Operators.NONE,Relation.OTHER,ParameterPriority.NONE
    );
    component3.getParameters().add(component3Parameter0);
    ParameterConfiguration component3Parameter1 = new ParameterConfiguration("component.outputType",
        "STRING",ParameterType.OTHER,Operators.NONE,Relation.OTHER,ParameterPriority.NONE
    );
```

Figure 6.14: Generated code for a configuration 1/3

quently, the graphical editor is also equipped with a code generation utility, using which the developers can generate the code for their configurations created using the graphical editor with a single clock of a button. This is highlighted in Figure 6.14, Figure 6.15 and Figure 6.16. These figures show the generated code for the configuration shown in Figure 6.12. The generated code for the configuration shows that it contains boiler plate code for defining components, their parameters, their ports and links between the components. In addition to the code generation, the graphical editor also performs configuration validation i.e. when the developer generate the code for a configuration, the code generation utility validates the configuration to make sure that all connections and parameters are set properly before the code is generated.

```
component3.getParameters().add(component3Parameter1);
PortConfiguration component3Port0 = new PortConfiguration("input", "INPUT","String");
component3.getPorts().add(component3Port0);
configuration.getComponents().add(component3);
ComponentConfiguration component4 = new ComponentConfiguration("ObjectToStringConverter",
    "mobi.narf.recognition.toolkit.generic.conversion.ObjectToStringConverter", false, false,"null");
ParameterConfiguration component4Parameter0 = new ParameterConfiguration("component.ignoreNull", "true",
    ParameterType.OTHER,Operators.NONE,Relation.OTHER,ParameterPriority.NONE
);
component4.getParameters().add(component4Parameter0);
ParameterConfiguration component4Parameter1 = new ParameterConfiguration("component.nullValue", "null",
    ParameterType.OTHER,Operators.NONE,Relation.OTHER,ParameterPriority.NONE
);
component4.getParameters().add(component4Parameter1);
PortConfiguration component4Port0 = new PortConfiguration("objectInput", "INPUT","Object");
component4.getPorts().add(component4Port0);
PortConfiguration component4Port1 = new PortConfiguration("outputString", "OUTPUT","String");
component4.getPorts().add(component4Port1);
configuration.getComponents().add(component4);
ComponentConfiguration component5 = new ComponentConfiguration("AudioSensor",
    "mobi.narf.recognition.toolkit.android.sensor.AudioSensor", false, true,
    "mobi.narf.recognition.toolkit.generic.folding.transformation.SensorTransformationComponent");
ParameterConfiguration component5Parameter0 = new ParameterConfiguration("component.audioSource", "1",
    ParameterType.OTHER,Operators.NONE,Relation.OTHER,ParameterPriority.NONE
);
component5.getParameters().add(component5Parameter0);
ParameterConfiguration component5Parameter1 = new ParameterConfiguration("component.frameNumber", "1",
    ParameterType.OTHER,Operators.NONE,Relation.OTHER,ParameterPriority.NONE
);
component5.getParameters().add(component5Parameter1);
ParameterConfiguration component5Parameter2 = new ParameterConfiguration("component.frameSize", "1024",
    ParameterType.WINDOW_SIZE,Operators.NONE,Relation.M_TO_N,ParameterPriority.PRIORITY_LOW
);
```

Figure 6.15: Generated code for a configuration 2/3

Graphical Editor Data Model

The data model for graphical editor is created using the Eclipse modelling framework and is shown in Figure 6.17. The model consists of six entities. The configuration entity represents configuration which in turn consist of other entities such as component and connection entities. The component entity in turn consists of port entity and parameter entity. The layout entity represents the graphical entity on which a configuration entity is drawn. Using the Eclipse modelling framework, we first created this model and by using its code generation utility, we generated the code of this data model and used it as input to the Eclipse graphical editing framework.

```
component5.getParameters().add(component5Parameter2);
ParameterConfiguration component5Parameter3 = new ParameterConfiguration("component.samplingDepth", "16",
    ParameterType.SAMPLING_DEPTH,Operators.NONE,Relation.M_IS_GRETER_THAN_N_TYPE,ParameterPriority.PRIORITY_MEDIUM
);
component5.getParameters().add(component5Parameter3);
ParameterConfiguration component5Parameter4 = new ParameterConfiguration("component.samplingRate", "8000",
    ParameterType.SAMPLING_RATE,Operators.NONE,Relation.M_TO_N,ParameterPriority.PRIORITY_HIGHEST
);
component5.getParameters().add(component5Parameter4);
ParameterConfiguration component5Parameter5 = new ParameterConfiguration("component.sleepTime", "0",
    ParameterType.OTHER,Operators.NONE,Relation.OTHER,ParameterPriority.NONE
);
component5.getParameters().add(component5Parameter5);
ParameterConfiguration component5Parameter6 = new ParameterConfiguration("component.stereo", "false",
    ParameterType.OTHER,Operators.NONE,Relation.OTHER,ParameterPriority.NONE
);
component5.getParameters().add(component5Parameter6);
PortConfiguration component5Port0 = new PortConfiguration("audioFrame", "OUTPUT","shortArray");
component5.getPorts().add(component5Port0);
configuration.getComponents().add(component5);
// create link configurations
LinkConfiguration link0 = new LinkConfiguration("ShortToDoubleFramer", "doubleSignalFrame", "FastFourierTransform", "signalFrame");
configuration.getLinks().add(link0);
LinkConfiguration link1 = new LinkConfiguration("FastFourierTransform", "fourierFrame", "SpectralEnergy", "inputFrame");
configuration.getLinks().add(link1);
LinkConfiguration link2 = new LinkConfiguration("SpectralEnergy", "energyValue", "ObjectToStringConverter", "objectInput");
configuration.getLinks().add(link2);
LinkConfiguration link3 = new LinkConfiguration("ObjectToStringConverter", "outputString", "IntentBroadcasterActivation", "input");
configuration.getLinks().add(link3);
LinkConfiguration link4 = new LinkConfiguration("AudioSensor", "audioFrame", "ShortToDoubleFramer", "shortSignalFrame");
configuration.getLinks().add(link4);
return configuration;
```

Figure 6.16: Generated code for a configuration 3/3

Graphical Editor Visual Model

The data model created using the Eclipse modelling framework is used in the Eclipse graphical editing framework for creating the graphical editor for the component system. In order to achieve that, we implemented a number of classes related to the different entities of the model. The entities for which the classes were implemented include component, configuration, connections, parameters and ports. In particular, we implemented EditParts for each entity. The EditParts are necessary for every entity that needs to be shown on the graphical pane. Similarly, we created Factory classes for these entities to instantiate them. We also created Figures for the entities which represents these entities visually. In order to enable interaction between developer and the visual representation of the entities, we implemented EditPolicies and different commands related to the policies such as create entity, delete entity, rename entity and resize entity.

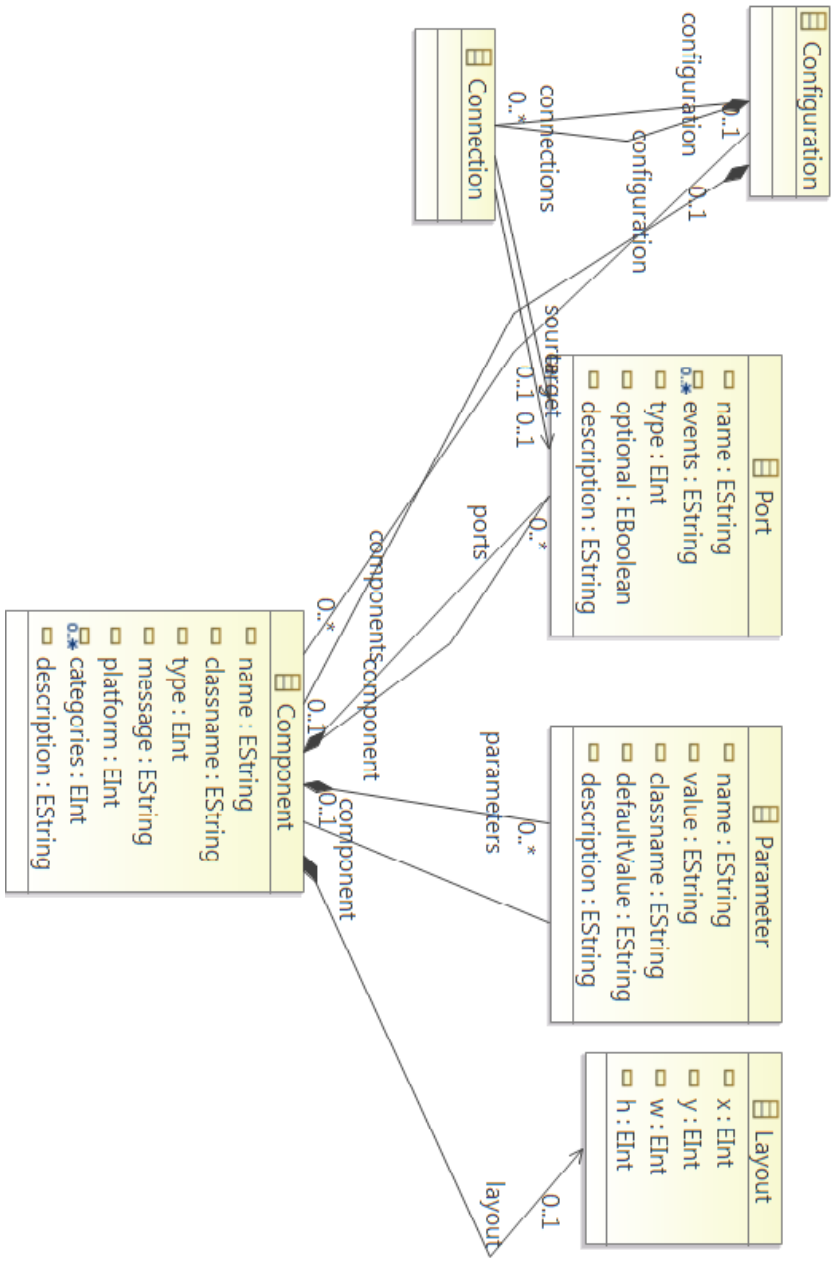

Figure 6.17: Eclipse modelling framework model of the component system

6.3.2 Component Toolkit

The component toolkit contains a large number of components that cover all levels of a typical context recognition application. At the sampling level, our toolkit provides components that access all sensor types available on Android based smart phones. This includes physical sensors such as accelerometers, microphones, magnetometers, GPS as well as WiFi and Bluetooth scanning components. In addition, we provide components to access virtual sensing sources such as on-line personal calendars, to-do lists. For preprocessing level, the toolkit contains various components for signal processing and statistical analysis. This includes simple components that compute averages, percentiles, variances, entropies, etc. over data frames as well as more complicated components such as finite impulse response filters, Fast Fourier transformations, etc. Furthermore, the toolkit also contains a number of specialized feature extraction components that compute features for different types of sensors such as the spectral rolloff and entropy or zero crossing rate which are used in audio recognition applications or WiFi fingerprints which can be used for indoor localization. At the classification layer, the toolkit contains a number of trained classifiers which we created as part of the audio, physical activity and location recognition applications.

Furthermore, there are a number of platform-specific components which are used to forward context to an application which enables the development of platform-independent classifiers. On Android based devices, for example, a developer can attach the output of a classifier to a broadcast component which sends results to interested applications using broadcast intents. Besides from components that are useful for applications, we have also developed a number of components that are useful for application development and performance evaluation. These includes components that record raw data streams coming from sensors as well as pseudo sensors that generate readings using pre-recorded data streams. Together, these components greatly simplify the application development process on mobile devices as they enable the emulation of sensors that might not be available on a development machine.

6.3.3 Component Selection

The rationale for the existing set of components in our component toolkit lies in the context recognition applications that we have developed during past years. The developed applications mainly targeted audio based [IHW+12b], location based [AHIM14] and physical activity [IFW+13] based context recognition applications for Android based smart phones. As a result, most of the components in our toolkit are targeted towards these contexts. The component toolkit not only provides set of specialized components for these target context but it also provides set of components which can be used for other contexts. To mention, the specialized components can also be used for other contexts depending upon the context recognition requirements. The component list also contains components required only during the training of the context recognition applications. These components mainly provide recording and replaying of sensor values to enable better analysis of raw data. The context recognition logics implemented by the specialized and other components

in our toolkit have been developed based on the literature survey we performed on context recognition applications and systems discussed in detail in the chapter on the related work. It might be possible that the same contexts can be recognized using other logical functions. It is also worth mentioning that the list of components in our toolkit is not complete because a developer might require components with functional requirements not fulfilled by the existing components in the toolkit. Hence, depending upon the context recognition requirements, more specialized and generic components can be added. In the following, we briefly discuss some of the components along with their UML diagrams that have been developed for audio recognition, location recognition and activity recognition applications. In addition, we also discuss components which can be used in the training phase of applications and for applications targeting other contexts.

6.3.4 Audio Components

The audio based context recognition applications require specialized components for the processing of audio data. In this regard, we have created different components to develop speech and music detection applications.

- LowEnergyFrameRate: This component calculates the percentage of frames whose root mean square (rms) value lies below the given threshold. The threshold is defined as a percentage of the root mean squared value.

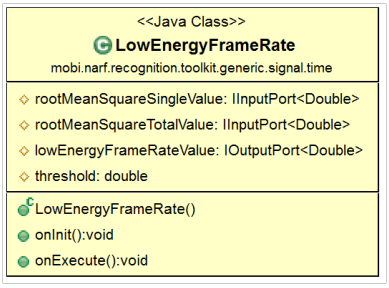

Figure 6.18: UML diagram of the LowEnergyFrameRate component

- ValueCrossingRate: This component computes the value crossing rate over each frame in the input buffer. This component can be used to mimic a zero crossing rate by setting the parameter value to 0.

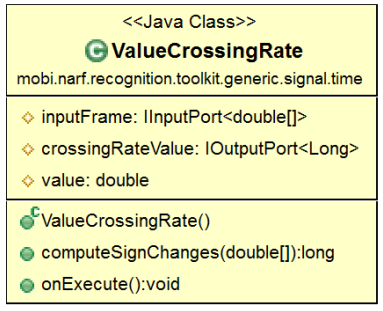

Figure 6.19: UML diagram of the ValueCrossingRate component

- AudioSensor: This component samples the device's microphone and returns it in constant sized frames that contain pcm encoded data. The component exposes different parameters to configure the audio sampling. The exposed parameters include frame size (the size of frames generated by the sensor), frame number (the number of consecutive frames to capture in one sampling cycle), sampling rate (the sampling rate in Hertz. Android devices are only guaranteed to support 44100 and 8000), sampling depth (the number of bits per sample. Currently, only 8 and 16 bits are supported).

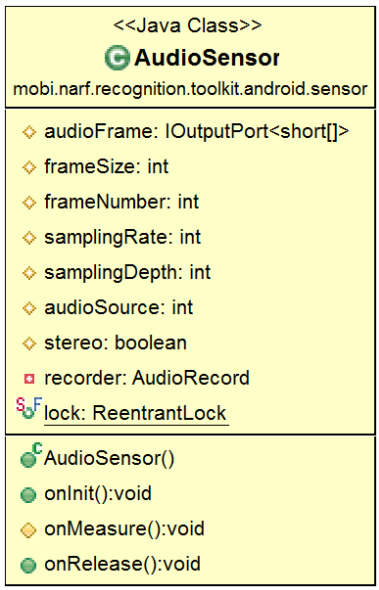

Figure 6.20: UML diagram of the AudioSensor component for Android

- FastFourierTransform: This component calculates the Fast Fourier Transform of a frame of time domain data. The input data has to be a power of two.

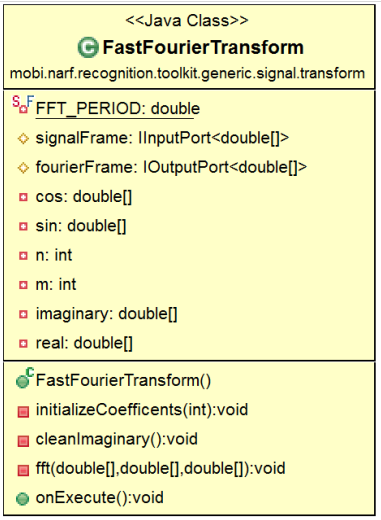

Figure 6.21: UML diagram of the FastFourierTransform component

- Bandwidth: This component calculates the bandwidth over the incoming frame using the spectral centroid value of the frame. One of the input ports receives the frequency domain values (array of double values) of the input frame and another port receives the spectral centroid.

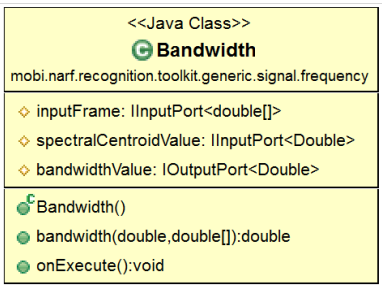

Figure 6.22: UML diagram of the Bandwidth component

- InformationEntropy: This component calculates the entropy of the information content of a given set of discrete signals. It calculates the weighed information score for each value in the input, the negative value of their summation is returned as the information entropy. The formula for information entropy is given as the product of the information content and the expected value(The negation of the sum of the product of the probability of a discrete value and the log base 2 of that value).

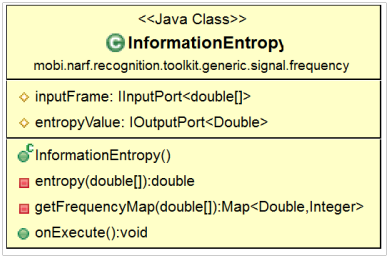

Figure 6.23: UML diagram of the InformationEntropy component

- SpectralCentroid: This component calculates the spectral centroid given a particular frequency domain frame as input. The spectral centroid can be thought of as the balancing point of a frequency spectrum.

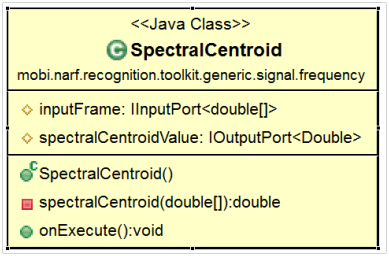

Figure 6.24: UML diagram of the SpectralCentroid component

- SpectralEnergy: This component calculates spectral energy of a signal. The energy content is given as a squared sum of the spectral coefficients.

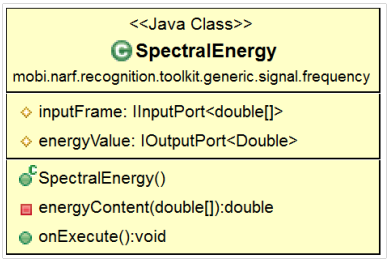

Figure 6.25: UML diagram of the SpectralEnergy component

- SpectralFlux: This component measures the change in the shape of the spectrum between the consecutive input frames. For this, it must slide over the input frames. Consequently, the input size is 2 and the flush value is 1. Thus, the output will be delayed accordingly by 1 frame.

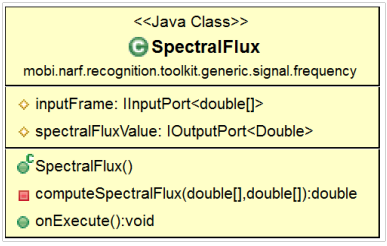

Figure 6.26: UML diagram of the SpectralFlux component

- SpectralRolloff: This component determines the frequency bin below which about 93 percent of the frequency distribution is concentrated.

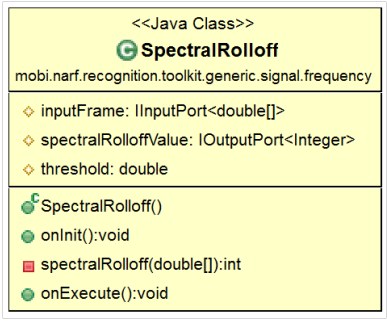

Figure 6.27: UML diagram of the SpectralRolloff component

6.3.5 Location Components

The location based context recognition applications require specialized components for the processing of location data. In this regard, we have created different components which we used in our location estimation and location prediction applications that we developed as part of EC projects GAMBAS and Living++. These components provide functionalities specific to indoor and outdoor localizations using some of the most commonly used localization techniques.

- ContinuousWiFiSensor: This component continuously scans for WiFi networks and delivers the results to the output port. The purpose of this component is to deliver the freshest values for the WiFi scans. The component keeps the scan results in a queue and flushes the queue if it becomes stale.

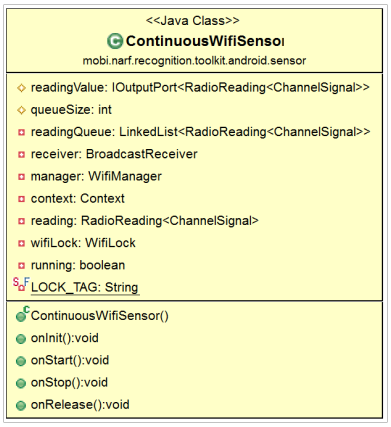

Figure 6.28: UML diagram of the ContinuousWiFiSensor component for Android

- GpsSensor: This component generates GPS readings in terms of longitude and latitude. The component is usually used for outdoor localization applications.

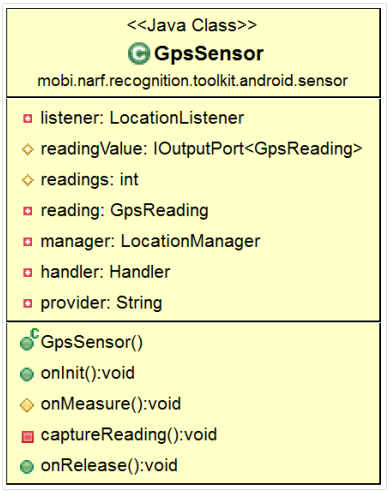

Figure 6.29: UML diagram of the GPSSensor component for Android

- GsmSensor: This component scans and reports the visible GSM cells. The number of visible cell are dependent on device's hardware. For some devices the numbers of visible cells are up to 6 whereas for others it could be only one. This component can be used for outdoor as well as indoor localization applications.

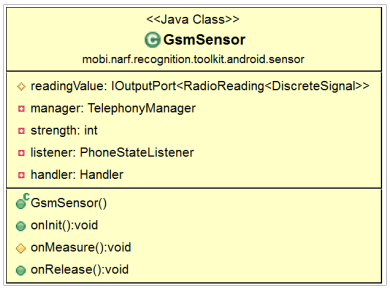

Figure 6.30: UML diagram of the GSMSensor component for Android

- LocationRetreiver: This component retrieves location information from the Location data type that we used in our localization applications. The Location data type represents a point in space and determines location in terms of x,y,z coordinates, latitude, longitude and altitude. The input to this component is a Location data type value and the output of the component is GPS coordinates of that location.

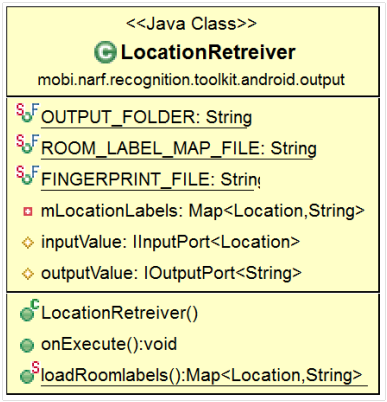

Figure 6.31: UML diagram of the LocationRetriever component for Android

- AccelerometerDeadReckoning: This component implements dead-reckoning and it predicts the next location estimate using the speed of motion derived from the accelerometer data.

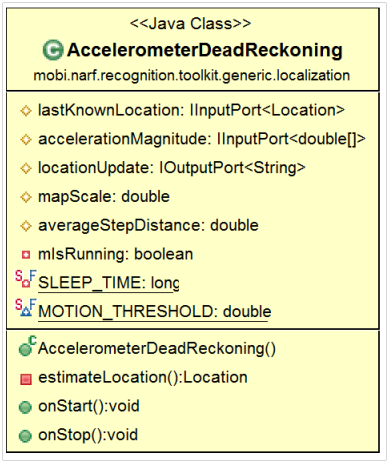

Figure 6.32: UML diagram of the AccelerometerDeadReckoning component

- RadioReadingParser: This component parses a reading looking for specific markers in the SSIDs and forwards recording readings if the marker is found. The markers for start, stop and collect recordings can be specified as parameters. If the start parameter is found, all readings will be forwarded to the output port until the stop marker is found. The collect marker triggers a system broadcast indicating that the collect marker has been found.

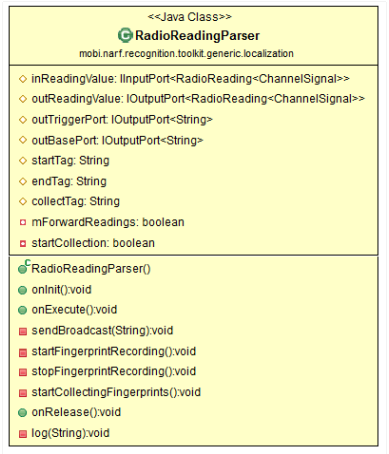

Figure 6.33: UML diagram of the RadioReadingParser component

- ReadingAnnotater: This component records scan readings together with the location coordinates where the scan was taken. The location coordinates are arbitrary and start from (0,0), and can be incremented by supplying events to the input port.

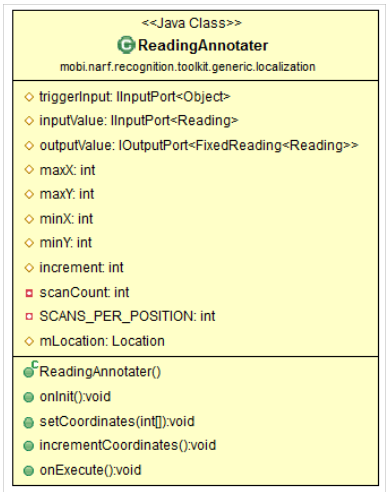

Figure 6.34: UML diagram of the ReadingAnnotater component

- ReadingTimeModifier: This component corrects the time stamps of a radio reading by a particular offset. If the offset is set to 0 the reading is just forwarded, otherwise a new one is created with the adjusted time stamp and the signals are copied into it.

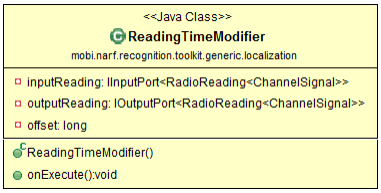

Figure 6.35: UML diagram of the ReadingTimeModifier component

6.3.6 Activity Components

The physical activity based context recognition applications require specialized components for the processing of data related to user's physical activities. In this regard, we created different components which we used in our fall detection, vital signs monitoring and mode of location determination applications that we developed as part of EC projects GAMBAS and Living++.

- GenericSensor: This is a generic sensing component which enables data sampling using all sensors provided by the android sensor manager. The component is called GenericSensor because using different parameters it can register with different sensors of the device e.g. if the sensorType is 1 then the component registers with accelerometer sensor, if sensorType is 2 then the component registers with magnetic field sensor, if the sensorType is 3 then the component registers with orientation etc. We use this component for the determination of physical movements.

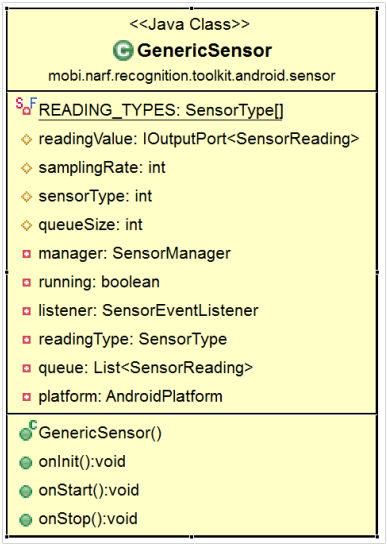

Figure 6.36: UML diagram of the GenericSensor component for Android

- CorrelationCoefficient: This component calculates the Pearson's product-moment coefficient, which measure the strength and direction of linear dependency between two set of values.

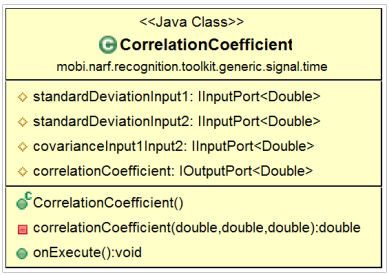

Figure 6.37: UML diagram of the CorrelationCoefficient component

- ArithmeticMean: This component calculates the arithmetic mean over all input frames.

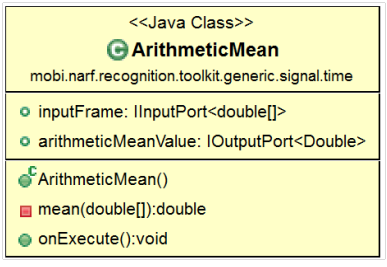

Figure 6.38: UML diagram of the ArithmeticMean component

- Covariance: This component calculates the covariance between two frames. Covariance is a measure of how two values change together between two set of discrete values. The input frames must be of equal size for this component.

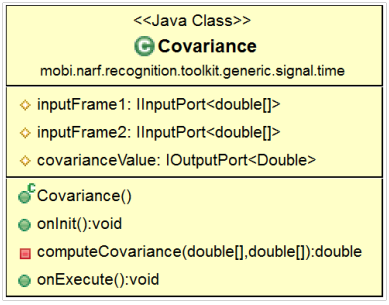

Figure 6.39: UML diagram of the Covariance component

- SignalMagnitudeArea: This component calculates signal magnitude area which is the area encompassed by the magnitude of each of the three axis of an accelerometer. For this component, the input frame sizes and frame numbers must be equal.

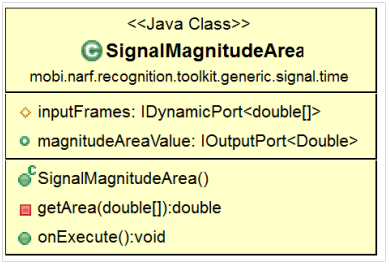

Figure 6.40: UML diagram of the SignalMagnitudeArea component

- SignalVectorMagnitude: This component calculates signal vector magnitude which is the magnitude of the 3 dimensional input frames. For this component, the input frame sizes and frame numbers must be equal.

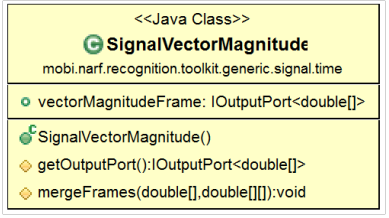

Figure 6.41: UML diagram of the SignalVectorMagnitude component

6.3.7 Miscellaneous Components

In addition to the specialized components for the location based, activity based and audio based context recognition applications, we have also created other components which can be used in applications targeting different contexts. We have also created components which can be used in the training phase of context recognition applications. In a typical training phase, the data from sensors is recorded and replayed in order to identify optimal set of processing and classification components.

- IntentBroadcaster: This component is used in Android based applications and broadcasts an intent [Andc] as a result of some string input. This component is usually the last component in a configuration. When a configuration completes a context recognition processing, this component sends the results to the application. Using parameters, it is possible to change the intent's action as well as the names of the extras that will contain the broadcaster's name as a string, the event's time (as a long in system milliseconds) and the event's data (as a string). The input port of this component receives an event and usually the event is the output of a classifier. The component then broadcast the intent using the intent filter defined through a parameter named action.

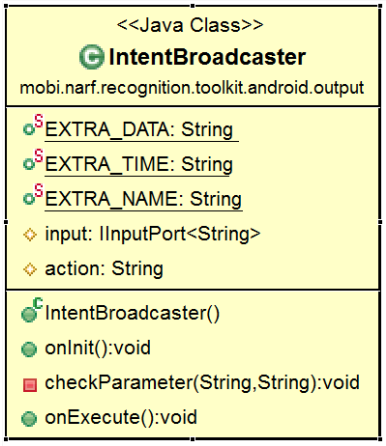

Figure 6.42: UML diagram of the IntentBroadcaster component for Android

- NotificationSoundPlayer: This component plays a beep sound on the Android device in response to any event. The event can be any context information e.g. movement of a user, reaching point of destination, entering in a meeting etc. The input port of the component receives the event as a trigger. Once the input is received, the component plays a notification sound using Android media player.

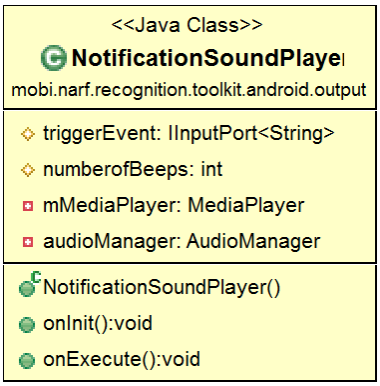

Figure 6.43: UML diagram of the NotificationSoundPlayer component for Android

- VibrationPlayer: This component enables vibration on the Android device in response to any event. Like NotificationSoundPlayer, the event can be any context information. The input port of the component is used to receive the event as a trigger. The component has no output port.

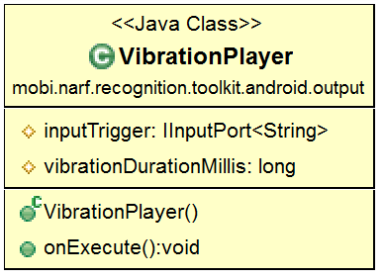

Figure 6.44: UML diagram of the VibrationPlayer component for Android

- BatterySensor: This component reports whether the device is charging or not and also returns the current battery state. This component sends the current battery level to its output port. The component computes the battery level by using the Android battery manger.

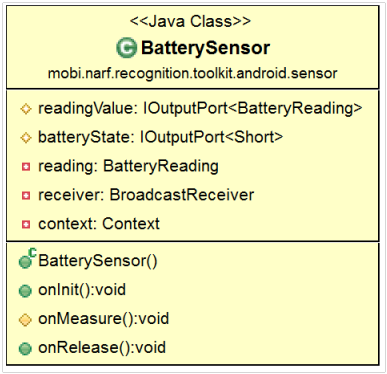

Figure 6.45: UML diagram of the BatterySensor component for Android

- TreeClassifier: This component implements a generic decision tree classifier that can read its model from the output of rapid miner. It is slower than hand-crafted classifiers but it is also more flexible and it avoids rewriting a lot of boiler plate if-else statements.

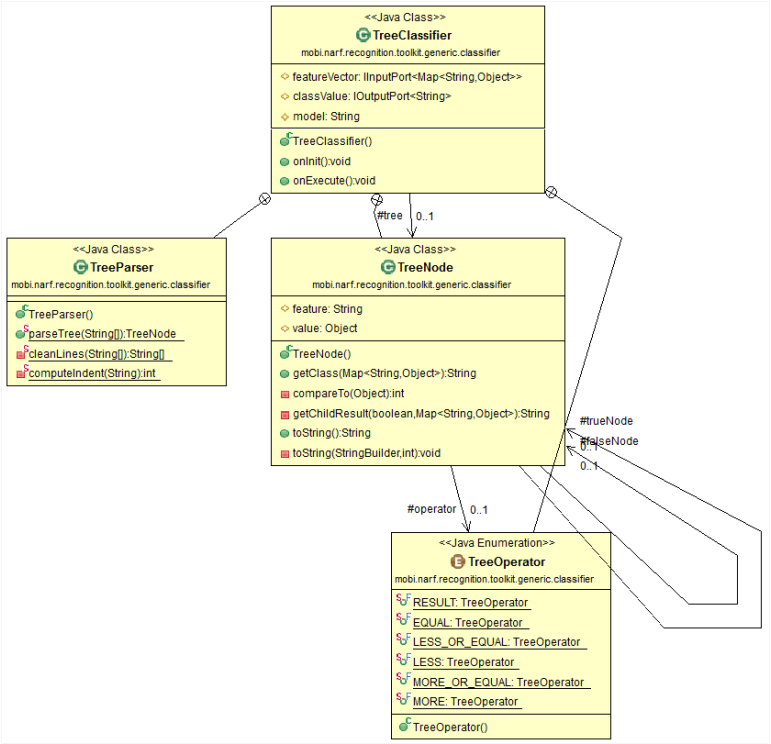

Figure 6.46: UML diagram of the TreeClassifier component

- MajorityClassifier: This component determines the class of context as the maximum of its inputs e.g. if the classification result of human body movement for last 10

recognition cycles contains 6 instance of walking and 4 instances of standing, then the majority classifier determines the current context as walking.

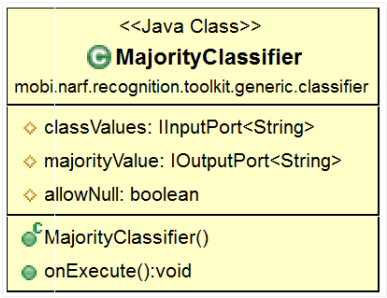

Figure 6.47: UML diagram of the MajorityClassifier component

- ShortToDoubleConverter: This component converts the incoming data in short format into a double data type. The frame size is not manipulated and in each execution, the complete buffered input is transformed, therefore, it is suggested to keep input flush and size equal to avoid duplicate conversions. The conversion components are necessary to provide a bridge between different components. For instance, if the microphone sensing component outputs data as array of short values and the available implementation of Fast Fourier Transform component (FFT) requires double values then a ShortToDoubleConverter conversion component can be used in between the microphone and the FFT component.

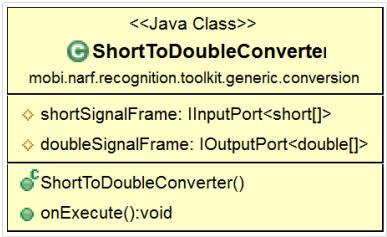

Figure 6.48: UML diagram of the ShortToDoubleConverter component

- ObjectToStringConverter: This component converts an input data object into a string. It is expected that the object knows how to write itself in an acceptable string format.

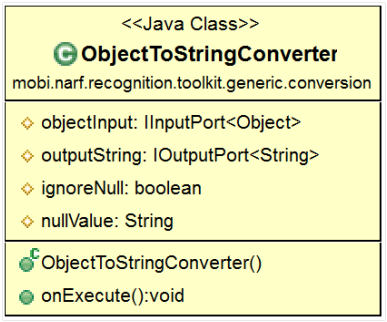

Figure 6.49: UML diagram of the ObjectToStringConverter component

- DoubleFramePlayer: This component emulates a set of sensor readings that have been written to a file. The sensors readings are emulated as double data types. The component can be used to replay the readings in the training phase of the classifier to determine the suitable set of pre processing functions.

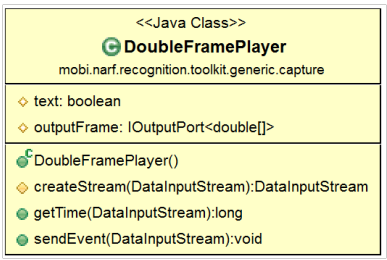

Figure 6.50: UML diagram of the DoubleFramePlayer component

- DoubleFrameRecorder: This component records double array event to disk such that they can be replayed with the double frame player. The input port of this component is used to receive the event to be recorded. For this component the input size of the event must equal the input flush value. Otherwise the same value would be recorded multiple times.

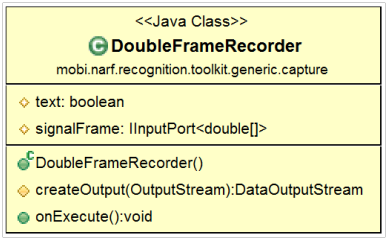

Figure 6.51: UML diagram of the DoubleFrameRecorder component

- RawAudioPlayer: This component reads a continuous raw audio stream and sends it to its output port. This component is similar to the DoubleFramePlayer but instead

of sending array of double values to its output, its sends raw audio values as array of shorts.

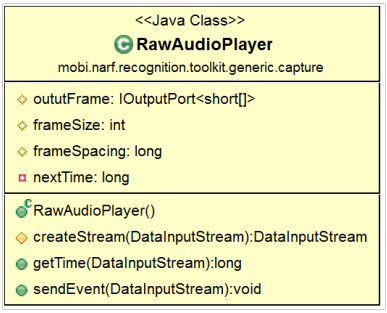

Figure 6.52: UML diagram of the RawAudioPlayer component

- RawAudioRecorder: This component records an incoming audio stream in raw format. The input port of this component is used to receiving raw audio stream to be recorded. The raw audio stream is passed to this component as array of short values.

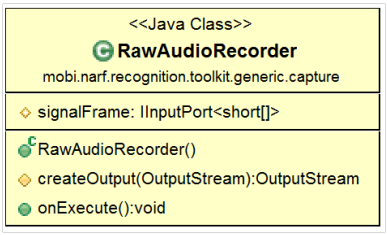

Figure 6.53: UML diagram of the RawAudioRecorder component

- Kurtosis: This component computes the kurtosis of a frame of double values. This is a measure of the skewedness of the distribution.

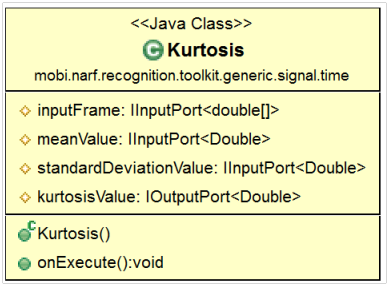

Figure 6.54: UML diagram of the Kurtosis component

- Percentile: This component calculates the percentile value over a single frame. This component can be used to mimic various statistical operators by setting the percentile value to a specific number.

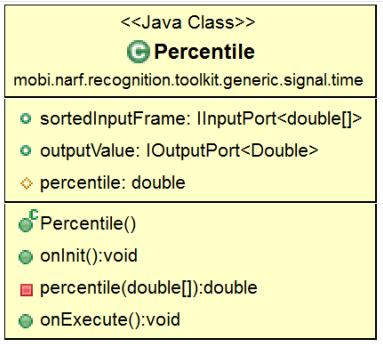

Figure 6.55: UML diagram of the Percentile component

- RootMeanSquare: This component calculates the root mean square over a frame. The root mean square of a frame is the square root of average of squared values in the frame.

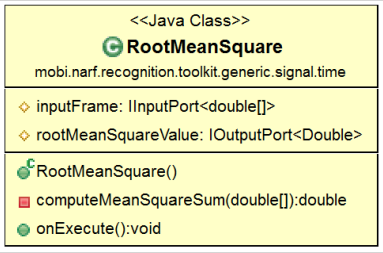

Figure 6.56: UML diagram of the RootMeanSquare component

- StandardDeviation: This component computes the standard deviation for a set of discrete signals.

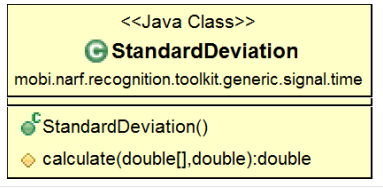

Figure 6.57: UML diagram of the StandardDeviation component

- ThresholdCrossingRate: This component computes how often the input buffer crosses the center value +/- the given threshold. A crossing is detected, if the last value was crossing the center value for at least the threshold value in the opposite direction.

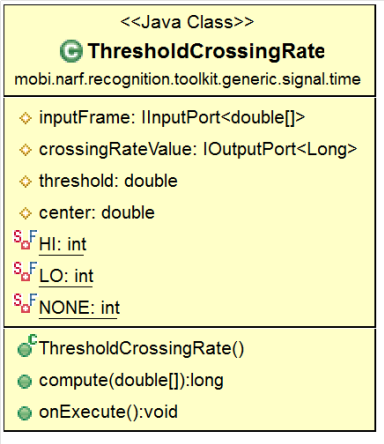

Figure 6.58: UML diagram of the ThresholdCrossingRate component

- Variance: This component computes the variance for a set of discrete signals. Variance determines how much deviation exists between values in a frame. Low value for variance indicates consistency of values.

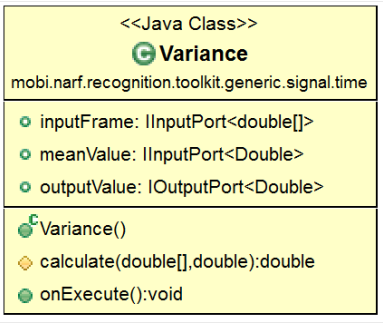

Figure 6.59: UML diagram of the Variance component

6.4 Evalutation

In this section, we discuss various applications that we have built using the component system. We also discuss how creation of these applications demonstrates the fulfilment of the design requirements of our framework. The component system has been extensively used in different European Commission (EC) projects namely GAMBAS[gam], Living++[liv], BESOS[bes] and SmartKye[sma]. In the following we describe its usage in GAMBAS and Living++ to demonstrate its usefulness in creating generic context recognition applications and prototypes as per the targeted use cases of these projects.

6.4.1 GAMBAS

GAMBAS (Generic Adaptive Middleware for Behaviour-Driven Autonomous Services) [AIP14] is a FP7 project from the EC. The goal of GAMBAS is to develop a middleware for personal mobile devices which enable development and execution of novel context recognition applications. As shown in Figure 6.60, our context recognition framework provides the data acquisition layer for the GAMBAS architecture. Thereby, our framework provides the basis of all the functionalities targeted by the applications. During the course of the project a number of applications and prototypes have been developed. The context recognition achieved by these applications and prototypes have been realized through the component system. The creation of configurations and the code generation for the configurations used in the applications have been done using the development tools of the component system. In the following we discuss some of these applications and prototypes.

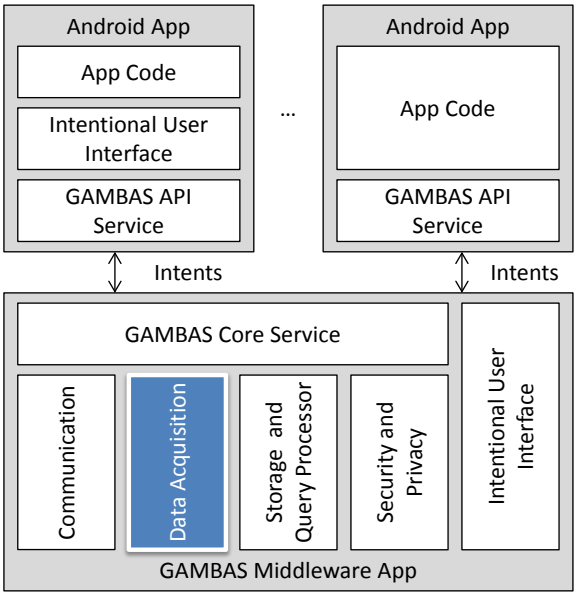

Figure 6.60: GAMBAS Architecture

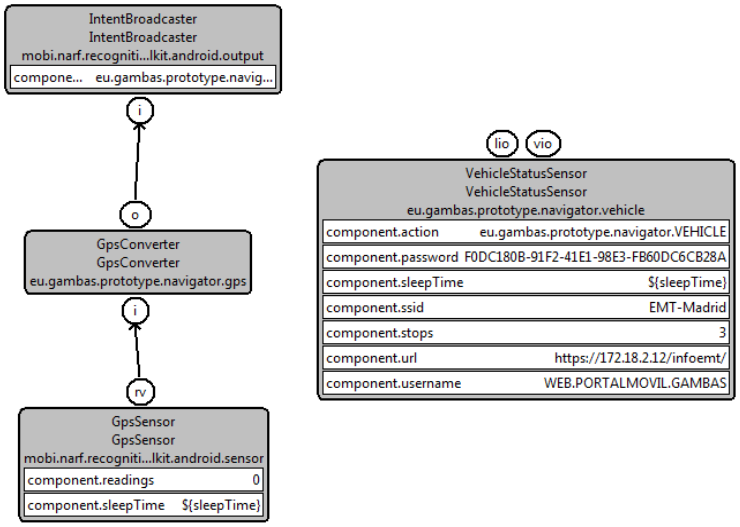

Figure 6.61: Configurations used in the Madrid navigator

Madrid Navigator

The Madrid navigator is the main prototype developed in GAMBAS and provides different bus navigation functionalities for public transport system in Madrid. These functionalities include calculation of routes, navigation support using maps and voice etc. The user's location acquisition required by the Madrid Navigator is done using the configurations created using the component system. The navigator acquires the location using GPS and/ or WiFi sensors of the phone.

Figure 6.61 shows the configurations used in the prototype. Figure 6.61 consists of two configurations. One configuration deals with acquisition and broadcasting of GPS information whereas the other configuration (consisting of one component) communicates with the web services running inside the buses through the bus WiFi connectivity. Through these web services the configuration reports different navigation related information to the application. This information includes current stop name, next stop name, time of arrival at the next stop etc. The loading of configurations, their instantiation and execution is handled by the runtime system of the component system. The configurations used in the prototype have been developed using the development tools of the component system. The use of graphical editor and code generation utilities highlights the support for rapid prototyping as depicted in Figure 6.61. Figure 6.62 shows the user interface of the Madrid navigator.

Bus Recognizer

The bus recognizer application recognizes whether the user in present on the bus. This is achieved by checking whether user's device can see a particular SSID for a long time. This application also handles various false positive cases such as determining user to be in the

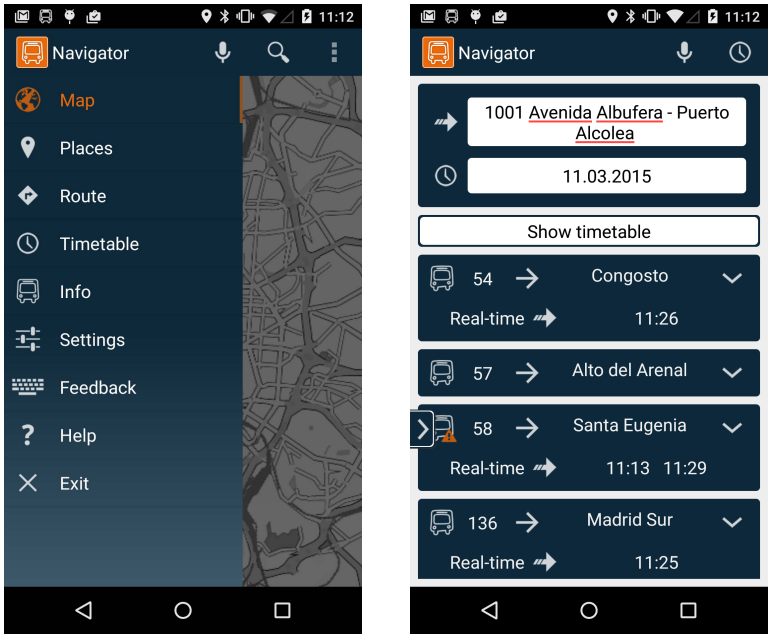

Figure 6.62: Screenshot of Madrid navigator

bus when the user is standing at the bus stop and a bus passes by or when the user is in the bus but the applications places the user in a nearby bus, for example in a traffic congestion situation when buses are moving closer to each other. The configurations used in the application uses WiFi and GPS sensing components along with components for uploading traces to a remote server to allow debugging and further optimizations of the classifier. The configurations used in the applications have been developed using the graphical editor of the component system. Figure 6.63 shows the screenshot of the application and the used configurations.

Weather Application

This application shows weather information for different cities for up to 3 days. The application demonstrates the integration of our framework with other modules of the GAMBAS middleware such as the semantic data storage and privacy preserving layers. The weather application consists of Android and J2SE components and uses weather data from [wet]. The J2SE based components from the component toolkit have been used to create services to retrieve and store weather data. Figure 6.64 shows the screenshot of the Android application and the J2SE components running on a backend server.

Location Predictor

The location prediction application predicts user's next intended location and also the duration of stay that user intends to stay at that location. The application uses three prediction techniques namely time series prediction, least k history predictor and location

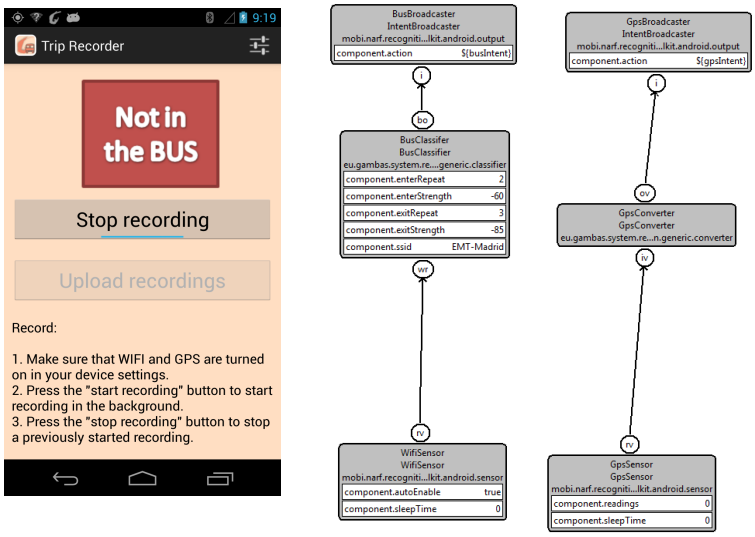

Figure 6.63: Configurations and the screenshot of the bus recognizer application

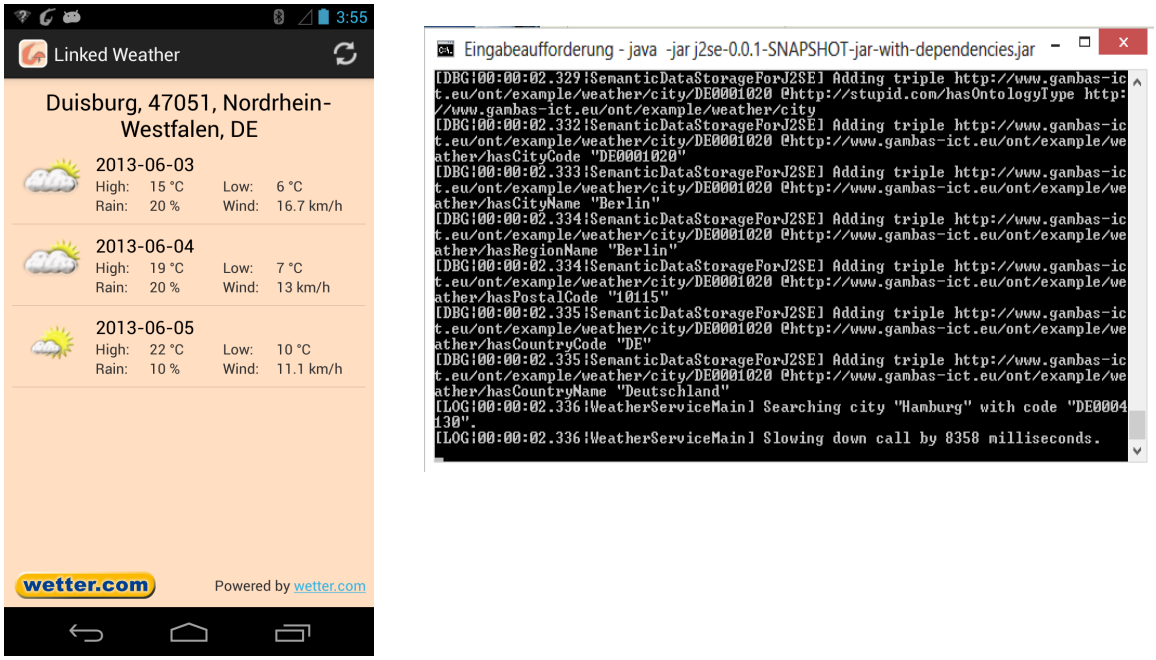

Figure 6.64: Screenshot of weather application (Android and J2SE components)

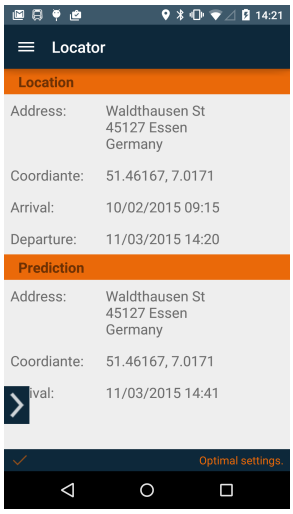

Figure 6.65: Screenshot of location prediction application

dependent Markov model. Figure 6.65 shows screenshot of the location prediction application. The application uses WiFi and GPS sensing components to acquire users' current location and then use this information to make the predictions.

Voiceprint Launcer

The voiceprint launcher application opens a particular application on a smart phone based on the spoken command. During the training phase, different voice commands are recorded and attached to different applications and then on runtime the voiceprint application maps the spoken command with a particular application based on the previous recordings. The voiceprint launcher application uses microphone sensing component from the component toolkit to enable microphone on the device. This application was developed as a prototype to demonstrate the effectiveness of speech recognition components which were later incorporated in the Madrid navigator application.

Emulator

The emulator application has been developed to emulate user trips so that they can be replayed for analysing, debugging and optimization of the Madrid navigator prototype. Since the Madrid navigator has been mostly developed in England and Germany but used in the city of Madrid, the emulator application provides an important tool for the remote analysis. The configuration used in the emulator application is shown in Figure 6.67. It can be seen that the configuration consists of a number of components such as ReadingPlayer for replaying the stored trace files, Reading synchronizer to synchronize readings from various sources, bus classifier for recognizing bus journey etc. The configuration also consists of broadcast components which are used to send different travel related information to the emulator application.

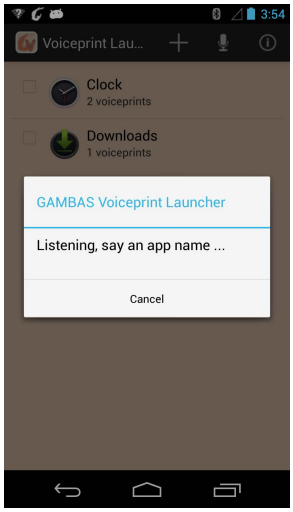

Figure 6.66: Screenshot of voice launcher application

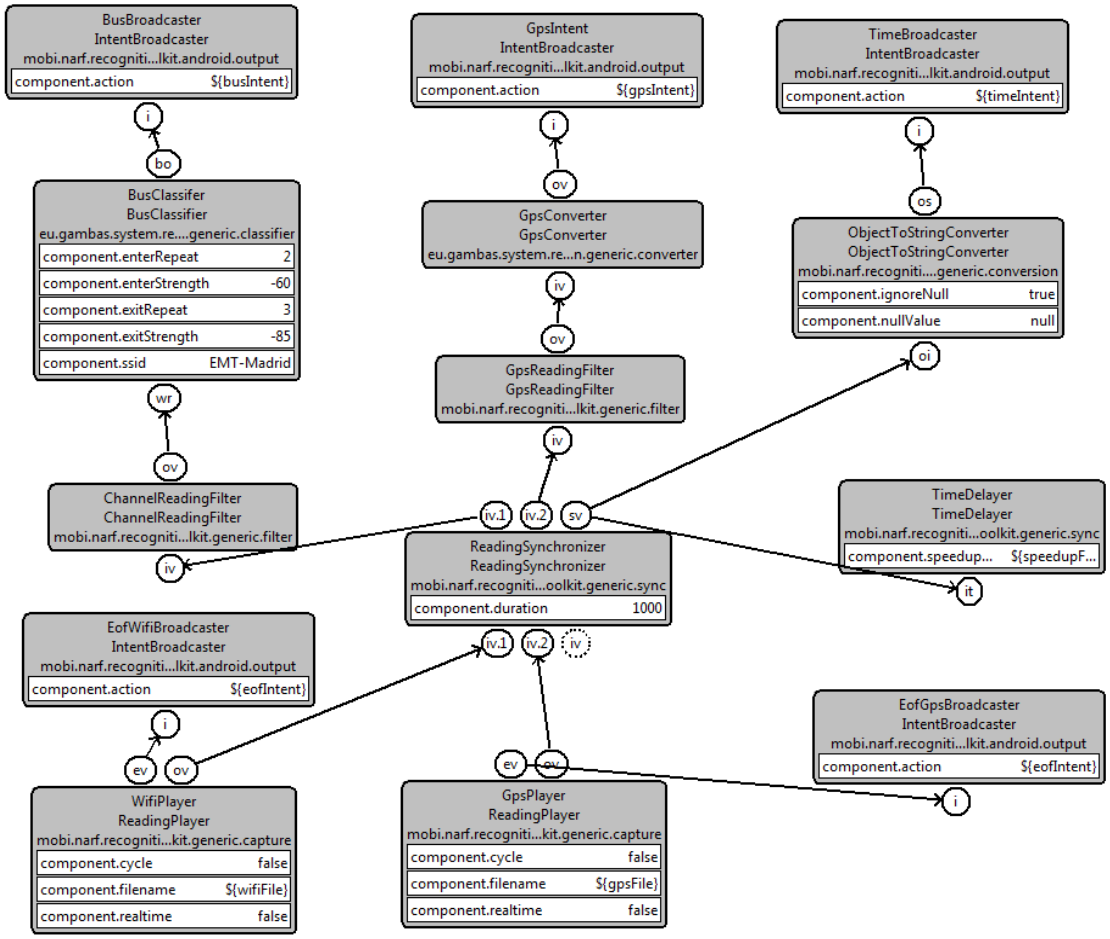

Figure 6.67: Configuration used for emulator application

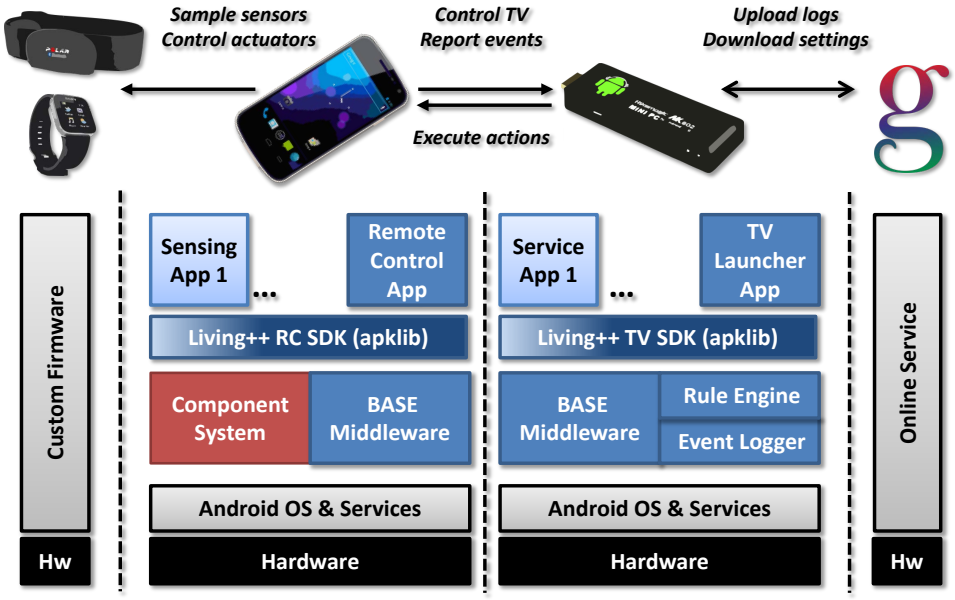

Figure 6.68: Living++ Architecture

6.4.2 Living++

Living++ (A Platform for assisted living applications) [IFW+13] is a EraSME project which provides assisted living support to the elderly people living alone. The project provides system support which enable care givers to remotely monitor their patients or the family members. The system consists of Android based hardware and software services which enable elderly people to live well and independently. The services available at the end of elderly persons mainly include health monitoring services. Our context recognition framework is a fundamental part of the overall architecture of Living++ platform as shown in Figure 6.68. The Living++ architecture provides a TV based health monitoring solution in which the users i.e. the elderly persons communicate with Android based smart TV via Andoird based smart phone using it as TV controlling as well as a communication device. The applications that we have built with Living++ span a broad spectrum ranging from providing convenience functionality to health related monitoring. Figure 6.69 and Figure 6.70 show some of these applications.

Weather

Weather is a simple TV application which shows the weather of different cities. The application is controlled through the remote control on the smart phone. Using left-right-ok and a keyboard that is displayed on the TV, a person can define places through which he can navigate using the up and down keys. The application is used by elderly people to check the weather conditions in order to dress accordingly. The screenshot is depicted in Figure 6.69.

Calendar

The calendar TV application displays the person's weekly agenda. The person can scroll through the different weeks using left-right buttons and scroll through different events using up-down buttons. Details can be displayed by pressing ok over a calendar item. The calendar app uses Google calendar as a backing data storage. The application is used by the elderly people to note the upcoming important events and reminders e.g. an upcoming birthday or a doctor's appointment. The screenshot is depicted in Figure 6.69.

Localization

The localization application is installed on the smart phone and provides indoor localization at room-level granularity with an update rate of approximately 0.5Hz. It relies on WLAN fingerprinting similar to RADAR [BP00] and incorporates the localization components from the component toolkit. The application keeps track of user's location and periodically send this information to TV based processing system. The system checks this information and alert userr or the care giver in case of any anomaly e.g. if a user has not come out of the bedroom for many hours then an alert is generated.

Activity and Vital Sign Monitoring

Similar to the indoor localization, these two applications consist of a set of components running on a smart phone report their measurements to the TV such that they can be used in rules to trigger an action/alarm or exported to Google drive for later retrieval and analysis. The activity monitoring applications uses the accelerometer of the phone to determine whether the person is standing, sitting or walking. Vital sign monitoring application integrates with a wearable pulse belt that reports its measurements to the smart phone through Bluetooth. The smart phone then forwards this information to the TV based processing system which checks the reported values and triggers an alarm in case the values are above or below certain thresholds.

Log Viewer

The log viewer application enables visual inspection of logged data produced by the localization and monitoring applications.

Context-aware Reminder

The Context-aware reminder is a TV application that is used to configure rules that issue reminders depending on the context of the user for example, it can be used to issue reminders only at certain places and at certain times.

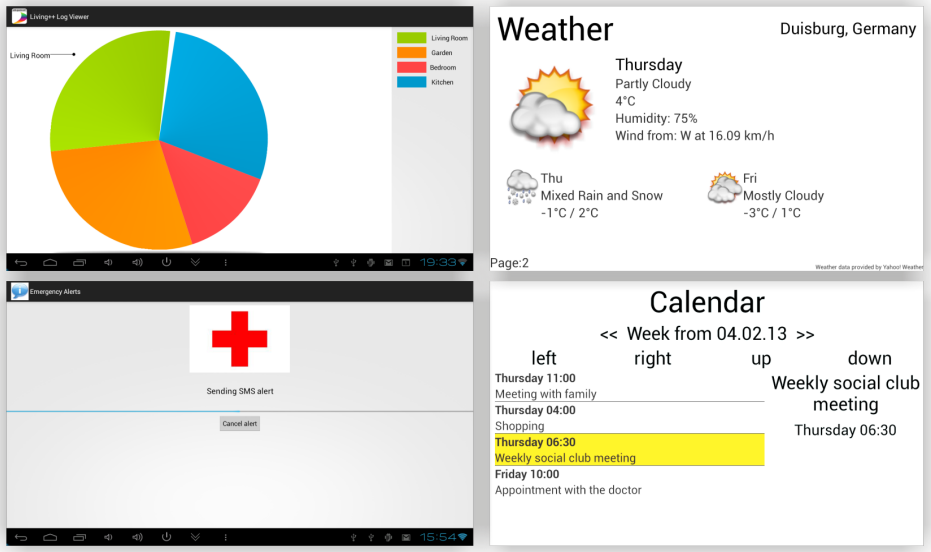

Figure 6.69: Screenshot of Living++ applications

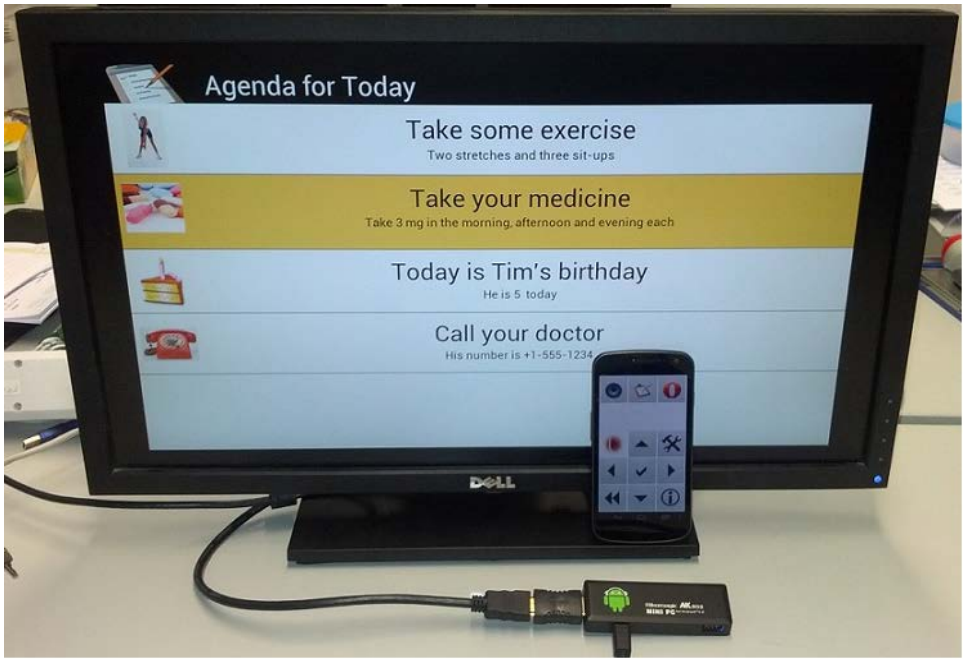

Figure 6.70: Screenshot of Living++ applications running on android enabled devices

Context-aware Alert

Context-aware alert application is used to define rules over the logs generated by other applications. By integrating with the vital sign monitoring application, for example, it is possible to generate emails or send SMS messages when the pulse of the elderly person exceeds a predefined threshold, for example.

Fall Detection

Fall detection can be seen as a special combination of activity monitoring and context-aware alert applications. However, instead of sending an asynchronous message, the TV application can establish a phone call using the smart phone of the user/patient, once a fall has been detected. To detect falls, we have implemented the approach described in [ST09].

6.4.3 Requirements Coverage

The applications and prototypes developed using the component system in GAMBAS and Living++ demonstrate the support for generic context recognition as it can be seen that the mentioned applications target a large variety of contexts such as location recognition, activity recognition, health monitoring etc. Below we take a look how these applications demonstrate the fulfilment of the four design requirements for our framework.

Uniformity

Uniformity is the ability to provide access to the context information required by different applications in a consistent way. The discussed applications cover a broad spectrum of contexts which shows that the component abstractions used by the component system enable the uniformity of design across different applications and as a result applications targeting different context information can access this information in a uniform manner. Specifically, as the context recognition logic of an application is represented by a configuration of components and connectors, the components relevant to different contexts are created in a same way independent of the context recognition logic they implement. The Madrid navigator, bus recognizer and emulator applications target different contexts but since these applications are created using the same component abstractions, the implementation of their context recognition logic follows the same design as can be seen from their configurations. Consequently, the application logic of these applications can access the results of the context recognition logic in a uniform manner.

Extensibility

Extensibility is the ability to extend the existing set of context recognition methods. In our component system this is possible at two level namely the application level and the system level. A the system the level, we have seen that the component system provides a

large component tool kit which offers already implemented components that can be used to create new applications without having the developers to implement them from scratch. At the application level, the extensibility is achieved through our component abstraction. As a configuration consists of components and connectors, it is trivial to extend an existing configuration for an application either by replacing components or by adding new components. The discussed applications can be extended to incorporate more functionalities e.g. the configuration of Madrid navigator application uses GPS configuration for determining the position of user but in case if the same applications needs to incorporate location determination through GSM cell tower, then this configuration can be extended by adding a component which determines location using GSM cell tower information. If this component is not already present in the toolkit, implementing it would also extend the toolkit, thereby exhibiting extensibility.

Configurability

Configurability is the ability to change the settings of the applications to enable different behaviour or different performance. In our component system, this is achieved through the parametrization support for the components. We can see that the configurations of the Madrid navigator, bus recognizer and emulator application consist of parametrizable components. For example, the VehicleStatusSensor component in the Madrid navigator application contains a parameter component.ssid which refers to the SSID of the access point used in a bus. If at any point in time the SSID of the access point is changed, this component parameter can be changed to reflect the new change without having to write the component again. Similarly, the WifiSensor component used in the configuration of bus recognizer application uses component.sleepTime variable to set the WiFi scanning interval. The value of this variable is set to zero which indicates that the WiFi scan is performed continuously. However, if at any point in time this interval needs to be extended, then this can be easily done by setting a new value for this variable. These examples show that the component abstraction used by our component system enables configurabiltiy.

Efficiency

The requirement on efficiency is described at two levels namely the efficiency for creating the context recognition applications and efficiency for executing these applications in an energy efficient manner. Here, we focus on efficiency for creating the applications. The off-line development tools provided by the component system enables rapid prototyping of context recognition applications. The tools contain a graphical editor and code generation utility to enable developers to create and test applications quickly. This is demonstrated through the applications discussed above. These development tools were used to visually draw the configurations and generate their code to use them directly in the applications. For example, consider the configuration used by the emulator application. This configuration consists of 14 different components connected with each other in different ways. The support for graphical editing enables developers to have a visual representation of this

complex configuration which enables them to modify the configuration using simple drag and drop operations without going through the code. Secondly, creating Java code for such a configuration is also time-consuming and prone to errors due to repetitive declaration of same constructs for components, parametrization, connections and ports. Therefore, the discussed applications show that the tool support provided by the component system enables efficiency for the developers.

6.5 Summary

This chapter has provided an in-depth detail about the component system. The chapter discussed different component abstractions namely component, connector and configuration in detail. The chapter also discussed the high level working of the component system. The chapter also showed various code snippets for components and configurations. In the latter part of the chapter, we discussed the off-line development tools of the component system. We discussed various aspects of the graphical editor and also provided a detailed overview of the component toolkit. We discussed a number of components which are part of the component toolkit. The chapter also discussed various context recognition applications that have been developed using the component system in different EU projects. The chapter concluded with a discussion on the coverage of design requirements by the component system. In the next chapter, we provide a detailed discussion on the second subsystem of our framework, the activation system.

7 ACTIVATION SYSTEM

In this chapter we discuss in detail the various aspects of the second system of our context recognition framework, the activation system. The activation system sits on top of the component system. The activation system uses the component abstractions provided by the component system to provide energy efficient execution of context recognition applications. Therefore, in order for the activation system to provide energy efficient execution of context recognition applications, the component system is also executed. The activation system provides a generic mechanism by which applications targeting any context type can enable their energy efficient executions by selecting only relevant set of configurations, thereby, fulfilling framework's requirement on energy efficiency. This means that if a context recognition application uses multiple configurations then instead of executing all of them at all the time, only those configurations are executed which are required at a particular time. In addition to that, the activation system also provides generic applicability of four energy efficiency techniques. These techniques include Suppression, Substitution, Piggybacking and Adaptation. These techniques have been explored for specific contexts in different existing systems. However, none of the existing systems provide their collective and generic applicability. With the activation system, it is possible for applications targeting any type of context to benefit from these techniques. The activation system enables the energy savings using a state machine abstraction. In this chapter, we discuss the state machine model used by the activation system in detail. We present the runtime system of the activation system and describe its building blocks. We discuss how activation system provides generic applicability of the four energy efficiency techniques. We also discuss the development tools associated with the activation system. Finally, we evaluate the activation system by creating a test application using its state machine model and by performing precise energy measurements on it to show the energy savings provided by the activation system.

7.1 Activation System Model

The activation system uses state machine abstraction for enabling energy efficient execution of context recognition applications. The state machine model used by the activation system consists of states and transitions. Conceptually, a state represents a step in a context recognition application at an arbitrary level of granularity. During this step, the application continuously recognizes certain context characteristics using a particular sensing and processing logic until a certain context constellation is detected. The constellations that

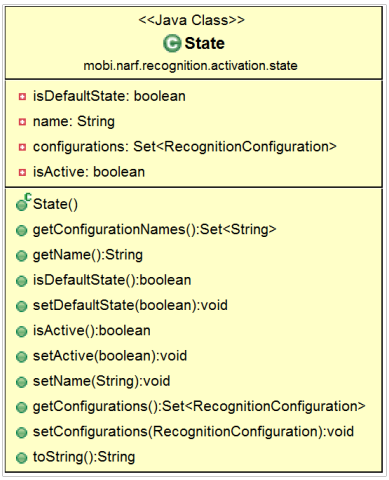

Figure 7.1: UML diagram of the state model

are relevant for the application are represented as transitions consisting of a source state, a target state and a rule, i.e. a set of conditions that describe the applicable constellations. When a constellation associated with a transition (whose source state is the current state) is detected, the state machine moves to the target state, thereby, effectively moving on to the next step in the application which may then modify or replace the sensing and processing logic. In the following, we discuss both states and transitions in more details.

7.1.1 State

A state represents a particular step in an application and is associated with a particular sensing and processing logic. To model this logic, states are associated with a collection of configurations which determine the relevant context characteristics. Depending upon the granularity of the step, the attached configurations could be detecting range of context characteristics from low level details such as movement of a user to high level details such as travelling in a high speed train or giving a presentation in a business meeting. In order to create a state, the developer must associate configuration(s) to it e.g. configuration for movement detection, configuration for meeting detection etc. To mention, the configurations associated with a state are the same configurations used by the component system described in the chapter on the component system. Hence, we can see that modelling of states and their execution is tightly coupled with the component system.

Figure 7.1 shows the UML diagram of the state model used by the activation system. A state has four member variables. The isDefaultState variable determines whether the state is the default state meaning that when a state machine is started the default state is the one which state machine executes first. In other words, the configurations associated with the default state are executed. There can only be one default state in the state machine. The name variable determines the identity of the state and it is unique among all the states. The isActive variable determines whether a state is executing or not and if it is not

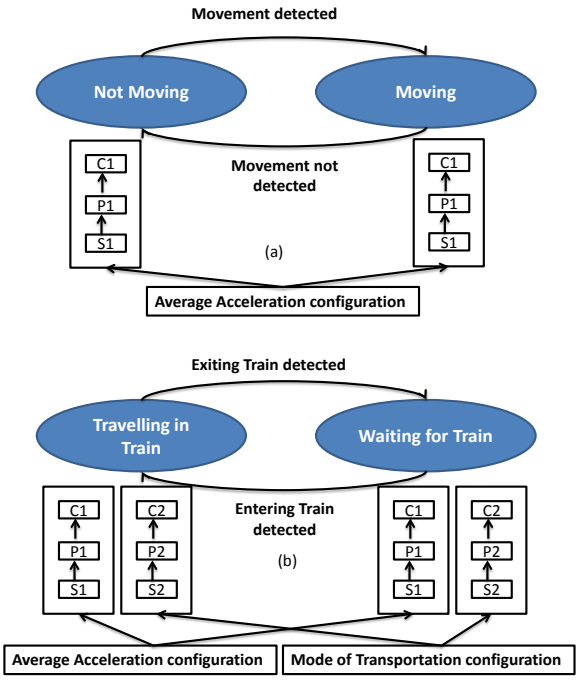

Figure 7.2: Example of state machine with different states

executing then the configurations associated it with are stopped. Finally, a state consists of a set of configurations which determine the current context. The member variable named configurations determines this set.

Examples of state machines with states with high and low levels context recognition are shown in Figure 7.2. Figure 7.2(a) depicts a state machine with two states. This state machine shows states which capture low level events. The two states capture user movement. If the user is not moving or moving less frequently, the state machine remains in Not Moving state. As soon as a transition is triggered, the state machine switches to the Moving state. In order to realize this state, the configuration for movement detection must be associated with both the states. Figure 7.2(b) depicts a state machine which switches between states to capture high level contexts. The two states capture user's mode of transportation. If the user is travelling in train, the state machine remains in the Travelling in Train state and as soon as the outgoing transition associated with this state is triggered, the state machine switches to the Waiting for Train state.

7.1.2 Transition

A transition represents a change between two states based on certain conditions. The conditions are the criteria which dictate whether the state machine should continue to execute the current state or not. A transition has one source state and one target state. When the conditions associated with a transition are evaluated to be true, the state machine leaves the source state and enters the target state. In the activation system, we realize the

conditions using rules. Each transition is associated with one rule but a rule may consist of one or more than one conditions. A condition is associated with one of the configurations attached to the state and it is modelled as abstract syntax tree to enable the formation of composed conditions, hence conditions represents a tree structure. Consequently, a condition consists of three nodes namely the operand, the operator and the value. If there are more than one conditions in a rule they are joined together with AND or OR clause. The transitions are evaluated using a rule engine and therefore, for a transition between states to take place it is necessary that the rule associated with the transition is evaluated to be true by the rule engine.

- Operand: An operand receives the outcome of the configuration associated with the condition. A rule is evaluated every time the operand receives any update.

- Operator: An operator represents a mathematical operator for evaluating the operand. The operators supported by our system include GREATER_ THAN (if operand is greater than the value), LESS_ THAN (if operand is less than the value) and EQUALS_ TO (if operand is equals to the value). A operand can be compared against an operator as long as there is a data type consistency e.g. if an operand represents a STRING then it is not possible to use it against GREATER_ THAN or LESS_ THAN operators.

- Value: A value is used to compare the output of the operand using the operator. A value can be either of NUMERIC data type or character based data type such as STRING. Similar to the case between operand and operator, the data type of value should be compatible with the data type of operand and operator e.g. if an operand reports a STRING and the operator is EQUALS_ TO but the values is NUMERIC then the operand cannot be compared against the value.

Figure 7.3 shows the UML diagram of transition model. The figure shows that a transition consists of a name for the source state and the target state. It also contains a rule which may consist of more than one condition. It also contains the name of the configurations that are associated with the source state to which the transition is attached. A transition has a priority. In case if a state has two or more outgoing transitions and if more than one transition are evaluated to be true at the same time then the transition which has a higher priority is executed and the state machine enters the target state of that transition. The outgoing transitions associated to a state have unique priorities.

Figure7.4 shows the UML diagram of the condition tree and related entities. This figure shows that a rule can have one condition tree and a condition tree may consist of many conditions. Each condition is modelled using three nodes. The figure shows that a node consists of five member variables. The three of them represent whether the node is operand, operator or the value node and this is determined by the node type. A node type consist of four enumerations namely operand, operator, value and join. The type join represents the joining node type such as AND or OR required to join two conditions.

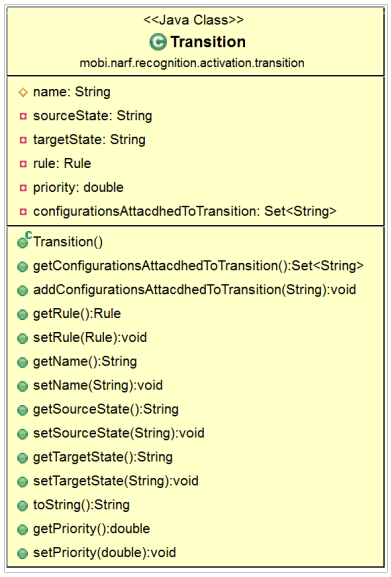

Figure 7.3: UML diagram of the transition model

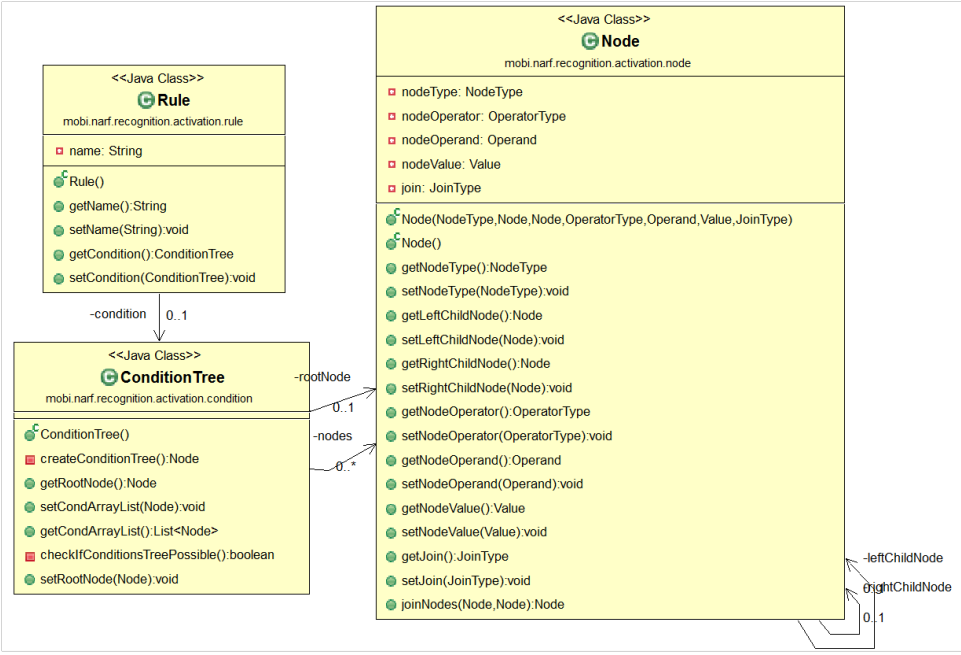

Figure 7.4: UML diagram of the condition model

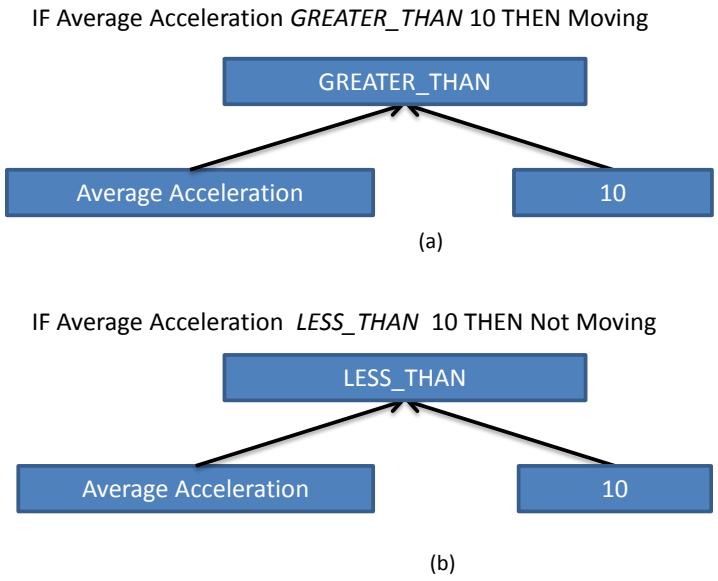

Figure 7.5: Example of a rule with one condition

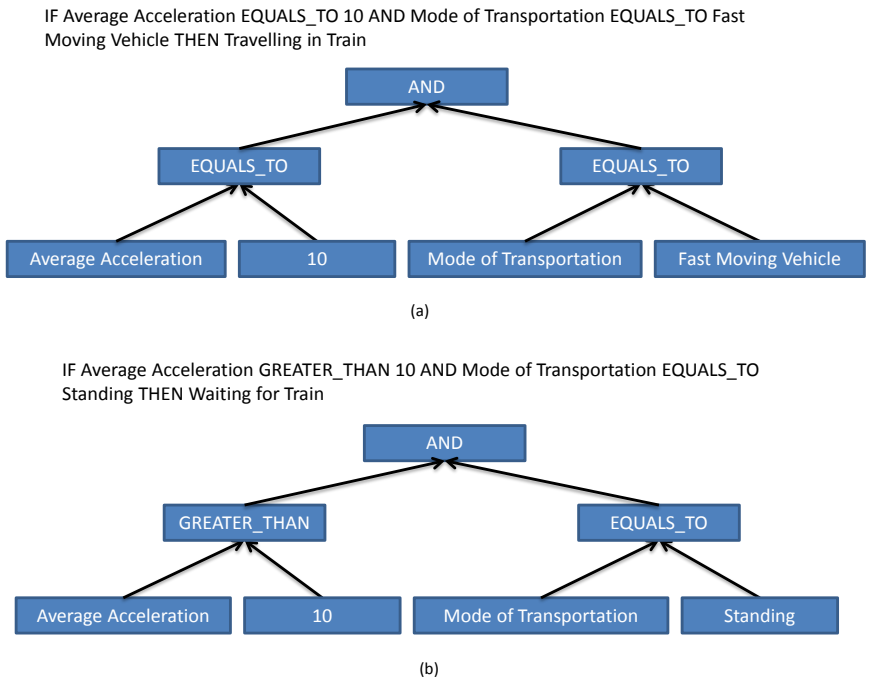

Figure 7.6: Example of a rule with two conditions

To further elaborate the structure of a transition, consider the example of rules associated with the transitions for the state machines shown in Figure 7.2 (a) and Figure 7.2(b) are shown in Figure 7.5 and Figure 7.6 respectively. The rule for Figure 7.2(a) has only

one configuration called "Average Acceleration" (which uses accelerometer data to determine user's movement) which is attached to both states "Moving" and "Not Moving". Subsequently, the same configuration is associated with the operand of the condition. The condition tree for the rules attached to the transitions of both states has three nodes. The rule for Figure 7.2(b) has two configurations "Average Acceleration" (which uses accelerometer data to determine acceleration and body posture of the user) and "Mode of Transportation" (which uses WiFi and sound information to determine the type of vehicle) attached to both states "Travelling in Train" and "Waiting for Train". The condition trees for rules attached to the transitions of both states have seven nodes, three nodes for representing two conditions and one node for the aggregation of the two conditions.

7.2 Activation System Execution

The main architectural building blocks of the activation system are shown in Figure 7.7. These include state machine, state machine instantiation logic and the rule engine. The applications can use activation system by creating a state machine using the development tools provided by the activation system. The state machine instantiation logic is responsible for identifying the current/default state in the state machine, communication with the component system and instantiation of rules. The rule engine process the rules attached to transitions based on the context information provided by the component system.

To demonstrate the interaction between the architectural building blocks shown in Figure 7.7, we briefly outline the general process depicted in Figure 7.7: (1) As the first step the application passes a state machine to the activation system. According to the state machine model described previously, a state machine consists of configurations attached to different states and rules associated with transitions between different states. (2) Once the state machine has been passed by the application, the activation system identifies the default state in the state machine and instantiates that state by sending the request to the underlying component system. The component system in turn instantiates the components and links described in the configurations attached to the state. Once the configurations are instantiated they start determining the context characteristics. Upon completion of each recognition cycle i.e. sampling of sensor data, processing of data and classification of processed data, the configuration sends the output to the activation system. (3) Next, the activation system instantiate the rules associated with the current state and then (4) activates the rule engine by passing rules to it. The rule engine evaluates the rules based on the output sent by the configurations and when a rule is evaluated to be true a change of state takes place. (5) When the need for a state change is detected, the activation system signals the change of state to the application. At the same time activation system signals this to the component system which stops the configurations attached to the current state and instantiates the configurations associated with the new state. Once this is done, the activation system instantiates the associated rules and the rule engine then starts evaluating the new rules using the new configurations and in this way the process continues.

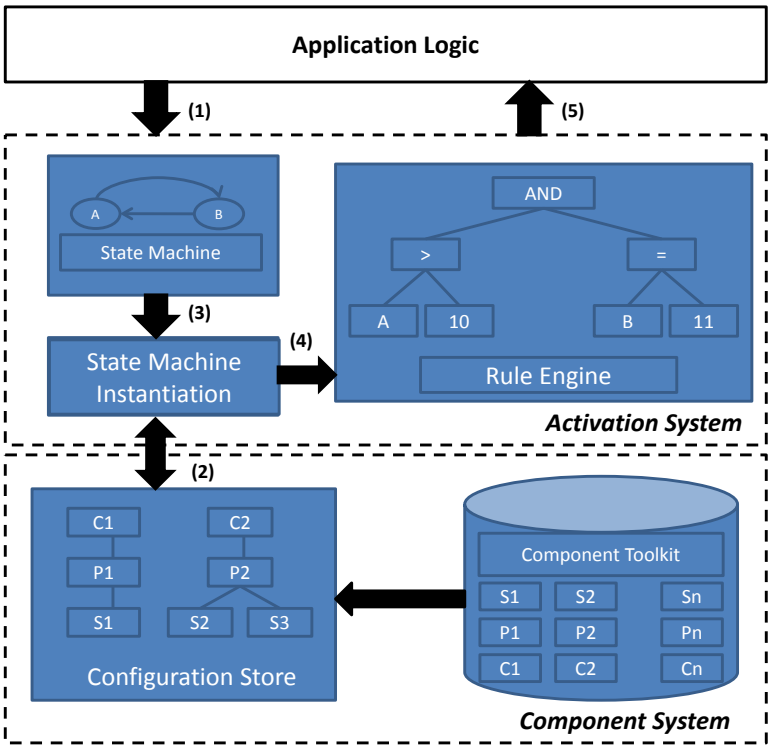

Figure 7.7: Architectural building blocks of activation system

7.2.1 Implementation

The activation system has been implemented for Android [anda] based devices. The activation system is implemented as an Android service such that it can be shared between different applications. Applications can start and stop state machines by sending their description as an Andorid Parcelable to the activation system service. The activation system service identifies the default state and if the Parcelable also contains a hierarchy of state machines it first identify the default state of the state machine in the highest level. Once the state is identified the activation system service starts another service that runs the component system and passes it the configurations attached to the state. The component system service performs configuration folding and instantiates the configurations. The outputs of running configurations are signaled via Android broadcasts. Once the configurations are started, the activation system service starts the rule engine service. The rule engine service registers the broadcast receivers for the currently executed configurations. When the broadcasts from the configurations are received, the rule engine evaluates the rules for all the outgoing transitions of the current state. If a state change occurs, the activation system service modifies the set of running configurations by starting and stopping them according to the state machine description. It then updates the rule engine and the registration of broadcast receivers accordingly. Finally, the change in states is sent to the applications via broadcasts so that the applications can perform any desired task.

7.2.2 Energy Efficiency Support

In Chapter 5, we discussed that one of the design rationale for using state machine abstraction for the activation system is due to the fact that with state machine abstraction we can provide the generic applicability of four energy efficiency techniques namely Suppression, Substitution, Adaptation and Piggybacking. In Chapter 5, we used a localization example to explain these techniques. In this section, we discuss how the activation system enables the four techniques to be used not just for location-based applications but for any type of context recognition application.

- *Suppression:* As any type of configuration (for determination of movement, for determination of sound etc.) can be attached to a state, it is completely flexible to use Suppression irrespective of type of target context. As authors of [ZKS10] used accelerometer to determine movement and then location, using our system it is possible to achieve the same by using configuration for accelerometer attached to one state and configuration for location to other state. Another example of Suppression could be to use audio sensor at low sampling rate to determine presence of some particular sound e.g. sound of opening of door or sound of cars first and then turn on GPS for outdoor location sensing. With activation system this example of Suppression can be realized by having two states, one with configuration for determining the sound and other for determining the location. Hence, with the state machine abstraction employed in our activation system, we can enable Suppression for different contexts.

- *Substitution:* Like Suppression the Substitution can also be applied in a generic way by the activation system. As Substitution means replacement of one sensing method with another, we can have different configurations for different sensing methods attached to different states. We can also have different state machines to achieve the same effect. With the evaluation of rules attached to transitions, the activation system can substitute one sensing method with other e.g. we can use one configuration for GSM localization attached to one state and second configuration for GPS localization attached to other state. Another example could be to use one configuration which detects whether the user is in a meeting based on his Google[goo] calendar entries and another configuration which detects whether the user is in a meeting based on the audio captured through the microphone of the phone.

- *Adaptation:* Just like Suppression and Substitution, Adaptation can conceptually be realized by introducing separate states for the different adaptation levels (e.g. one state for detecting movement with low precision if the phone's battery level is low and another state for detecting movement with high precision if the phone's battery level is high) and associating different configurations (e.g. one with a low sampling rate and one with a high sampling rate) with the states. However, since this will generally lead to complex state machines with a large number of transitions to represent the adaptation, it is typically more convenient to use separate state machines to represent the different levels and to introduce a hierarchy of state machines such that the state

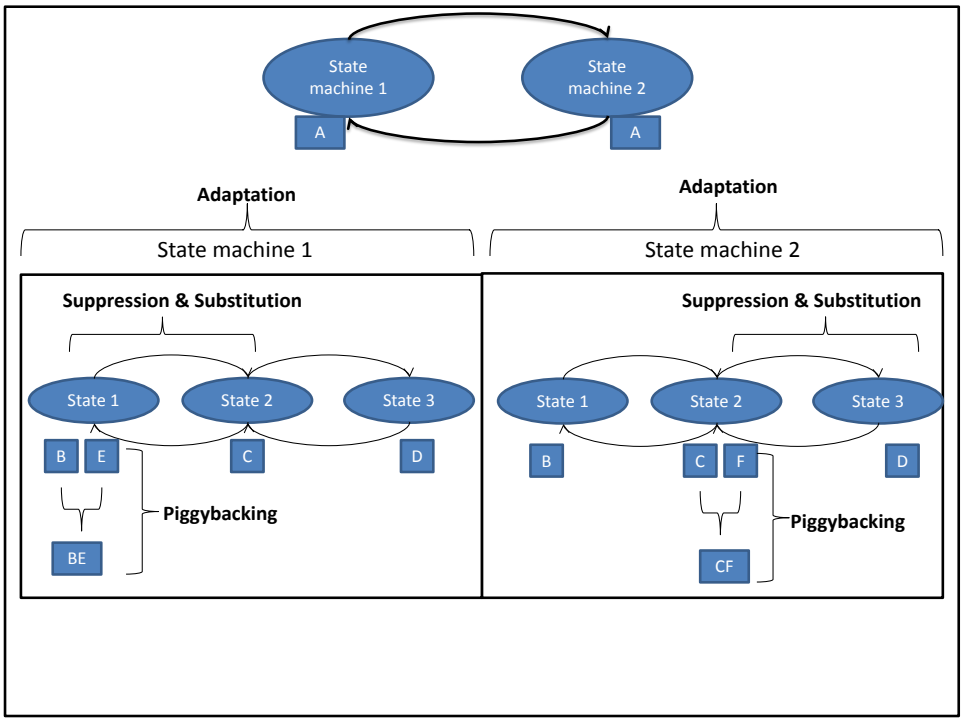

Figure 7.8: Generic applicability of four energy efficiency techniques

machine at the highest level is used to detect the appropriate adaptation level. This state machine is then responsible for activating the appropriate variant of the actual state machine that performs the recognition. In order to thoroughly support this, the activation system enables developers to directly associate state machines with states which technically enable a hierarchical composition. Yet, it is noteworthy to point out that this is primarily a tool for simplifying the development as it is possible to generate a single machine to handle adaptation.

- *Piggybacking:* As already mentioned in Chapter 5, the underlying component system that we use to run the sampling and processing logic for the activation system already provides configuration folding as a generic solution to achieve Piggybacking. Thus, if a state of a state machine exhibits multiple configurations or if multiple state machines are executed simultaneously, the configuration folding algorithm of the component system will ensure the energy efficient execution of the complete set of configurations that is running at the same time by removing redundant computations between them similar to Piggybacking. More details on configuration folding are provided in the next chapter.

The generic applicability of these four energy efficiency techniques is shown in Figure 7.8 which consist of a hierarchy of state machines. The state machine at the highest level is used to control the execution of two other state machines. Both the states in the state machine at the highest level have a variant of configuration A attached to it. Depending

upon the transition, either state machine 1 or state machine 2 is active at any time. This switching of state machines demonstrates application of Adaptation. In state machine 1 and state machine 2, there are three states and depending upon the definition of the states, the switching between states can either represent Suppression or Substitution. Lastly, State 1 and State 2 in the two state machines have two configurations attached to them. Thus, configuration folding is applied to these configurations in the two states to produce a single configuration which demonstrates the applicability of Piggybacking.

7.3 Development Tools

The activation system is equipped with set of off-line development tools to facilitate the development of state machines to be used in the applications. These tools include graphical editor for the creating the state machines and state machine validation and code generation utilities.

7.3.1 Graphical Editor

The graphical editor for the state machines enables rapid prototyping of state machines so that they can be developed quickly and used in the applications without having developers to write code themselves. The graphical editor for state machine is implemented as an Eclipse plug-in using the Eclipse modelling framework and Eclipse graphical Editing framework. The graphical editor for state machine provides following functionalities.

Graphical Editing

The graphical editor enables developers to draw state machines for their applications. As a state machine consists of set of states and transitions between states, this essentially means that the developers can create the state machines on the graphical editing pane using states and connect those using transitions. In order to create a state machine, the developers can drag and drop the states on the graphical pane, set their name, set the configurations to be attached to them. In addition, the developers can also create rules for the transitions. In order to create the rules, the developers can specify the conditions and set the operator, operand and values nodes. When developers create a transition between the two states, the graphical building block for the transition appears which is used for creating the rule associated with transition. Figure 7.9 shows an example state machine created using our graphical editor. The state machine consists of two states named Standing and Jogging. The names of the states, the default state and the configurations to be attaches to the states are set by the developer. The state Standing has a configuration Standing Conf attached to it while state Jogging has Jogging Conf attached to it. In this state machine, state Standing is the default state. The figure also shows transitions between the states. The transition from state Standing to state Jogging has one rule attached to it. This rule consists of one condition. The operand of the condition is the configuration Standing Conf, the operator is GREATER_THAN and the value is 10. In order for this transition to take place, the result returned by the configuration Standing Conf should have a value greater than value 10. Similarly, the transition from state Jogging to state Standing also has one rule with one condition. For this transition to take place, the result returned by the Jogging Conf should be less than value 10.

Code Generation for State Machines

Depending on the size of the state machine, the number of lines of code required to define a state machine description can be comparatively large. For example, a simple state machine

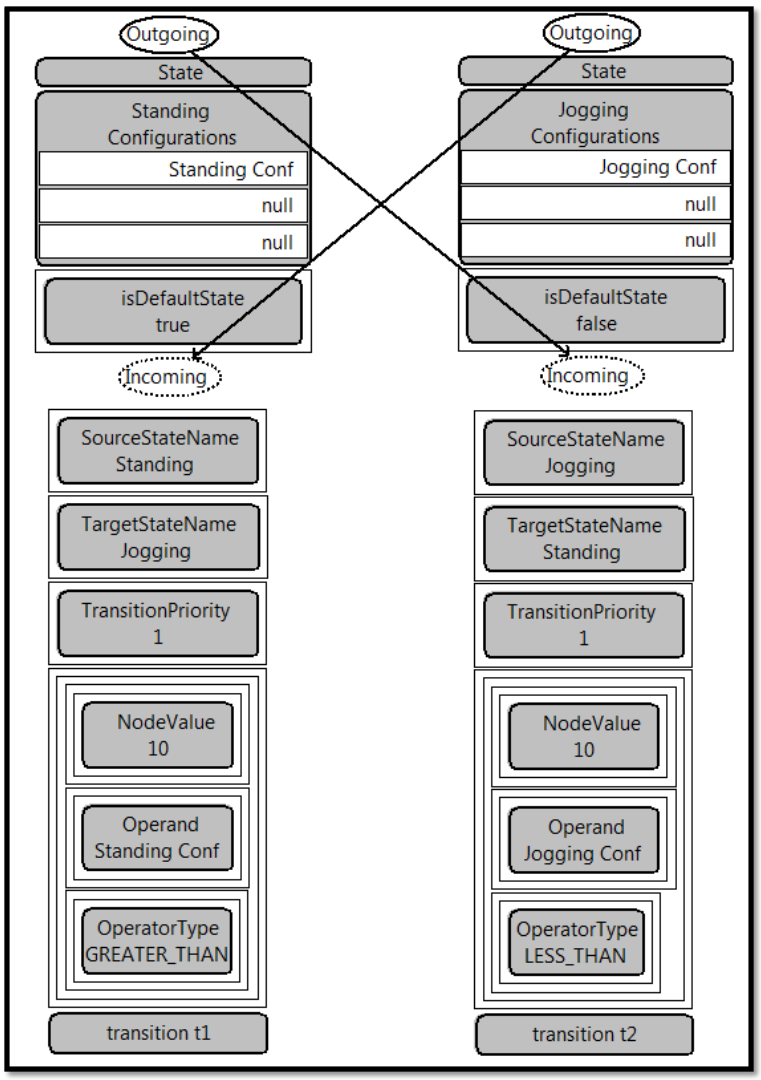

Figure 7.9: Screenshot of the graphical editor for activation system with two states

```java
public class FitnessStateMachine {

    /**
     * Creates a new state machine configuration with the specified name.
     *
     * @param configurationName The name of the state machine configuration to create.
     * @return A configuration with the specified name.
     */
    public StateMachineConfiguration createConfiguration(String configurationName) {
        StateMachineConfiguration configuration = new StateMachineConfiguration(configurationName);
        // create states configurations
        State state0 = new State();
        state0.setName("Standing ");
        state0.setDefaultState(true);
        Standing Conf Standing Conf0 = new Standing Conf();
        state0.setConfigurations(Standing Conf0.createConfiguration("Standing Conf"));
        configuration.setState(state0);


        State state1 = new State();
        state1.setName("Jogging ");
        state1.setDefaultState(false);
        Jogging Conf Jogging Conf1 = new Jogging Conf();
        state1.setConfigurations(Jogging Conf1.createConfiguration("Jogging Conf"));
        configuration.setState(state1);


        // create transition configurations
        Transition transition0 = new Transition();
        transition0.setName("transition t1");
        transition0.setPriority(1);
        transition0.setSourceState("Standing ");
        transition0.setTargetState("Jogging ");
        Rule rule0 = new Rule();
        rule0.setName("Rule");
        ConditionTree conditionTree0 = new ConditionTree();
```

Figure 7.10: Generated code snippet of a state machine 1/3

```
Value value0 = new Value();
value0.setValue("10");
//setting up operand
transition0.addConfigurationsAttacdhedToTransition("Standing Conf");
Operand operand0 = new Operand(Platform.ANDROID,"Standing Conf");
//setting up operator
Node node_operand0 = new Node(NodeType.OPERAND,null,null,null,operand0,null,null);
Node node_value0 = new Node(NodeType.VALUE,null,null,null,null,value0,null);
Node node_operator0= new Node(NodeType.OPERATOR,node_operand0,node_value0,OperatorType.GREA
//setting up nodes in the condition tree
conditionTree0.setCondArrayList(node_operand0);
conditionTree0.setCondArrayList(node_operator0);
conditionTree0.setCondArrayList(node_value0);
// setting up rule
rule0.setCondition(conditionTree0);// adding rule to transition
transition0.setRule(rule0);// adding transition to state machine
configuration.setTransition(transition0);

Transition transition1 = new Transition();
transition1.setName("transition t2");
transition1.setPriority(1);
transition1.setSourceState("Jogging ");
transition1.setTargetState("Standing ");
Rule rule1 = new Rule();
rule1.setName("Rule");
ConditionTree conditionTree1 = new ConditionTree();
//setting up value
Value value1 = new Value();
value1.setValue("10");
//setting up operand
transition1.addConfigurationsAttacdhedToTransition("Jogging Conf");
Operand operand1 = new Operand(Platform.ANDROID,"Jogging Conf");
```

Figure 7.11: Generated code snippet of a state machine 2/3

```
    //setting up operator
    Node node_operand1 = new Node(NodeType.OPERAND,null,null,null,operand1,null,null);
    Node node_value1 = new Node(NodeType.VALUE,null,null,null,null,value1,null);
    Node node_operator1= new Node(NodeType.OPERATOR,node_operand1,node_value1,OperatorType.LESS_THAN,null,null,null);
    //setting up nodes in the condition tree
    conditionTree1.setCondArrayList(node_operand1);
    conditionTree1.setCondArrayList(node_operator1);
    conditionTree1.setCondArrayList(node_value1);
    // setting up rule
    rule1.setCondition(conditionTree1);// adding rule to transition
    transition1.setRule(rule1);// adding transition to state machine
    configuration.setTransition(transition1);

    return configuration;
}
```

Figure 7.12: Generated code snippet of a state machine 3/3

consisting of two states with one configuration attached to each state and one outgoing transition per state requires about 75 lines of code. Although, this code is conceptually trivial but due to the the fact that it requires the repetitive and consistent use of different model elements such as states, transitions, nodes, operands, operators, values, etc., its manual creation is a tedious process that is prone to errors. Specially, during the training phase of the application when a developer creates different state machines to test their accuracies, writing the code manually is not an optimal approach. Consequently, the graphical editor is also equipped with a code generation utility, using which the developers can generate the code for their state machines created using the graphical editor with a single click of a button. This is highlighted in Figure 7.10, Figure 7.11 and Figure 7.12. These figures show the generated code for the configuration shown in Figure 7.9. In addition to the code generation, the graphical editor also performs configuration validation i.e. when the developer tries to generate the code for a state machine, the code generation utility validates it to make sure that all transitions and rules are set properly before generating the code.

Graphical Editor Data Model

The data model for graphical editor is created using the Eclipse modelling framework and is shown in Figure 7.13. The model consists of many entities. To mention, the model shown in this figure is the extension of the model used for the graphical editor of the component system. In this model the entities specific to the graphical editor of the activation system includes State, Transition, OperatorType, NodeType, Platform, NodeTypeClass, NodeOperatorClass, Node, ConditionTree, Rule, Taarget, Value, Priority and AttachedConfiguration. Using the Eclipse modelling framework, we first created this model and by using its code generation utility, we generated the code of this data model and use it as input to the Eclipse graphical editing framework.

Graphical Editor Visual Model

The data model created using the Eclipse modelling framework is used in the graphical editing framework for creating the graphical editor for the activation system. In order to achieve that, we implemented a number of classes related to the different entities of the model. The entities for which the classes were implemented include state, transition, rule, condition, node, operand, operator and value. In particular, we implemented EditParts for each entity. The EditParts are necessary for every entity that needs to be shown on the graphical pane. Similarly, we created Factory classes for these entities to instantiate them. We also created Figures for the entities which represents these entities visually. The example of these figures can be seen in Figure 7.9. In order to enable interaction between developer and the visual representation of the entities, we implemented EditPolicies and different commands related to these policies such as create entity, delete entity, rename entity, resize entity etc.

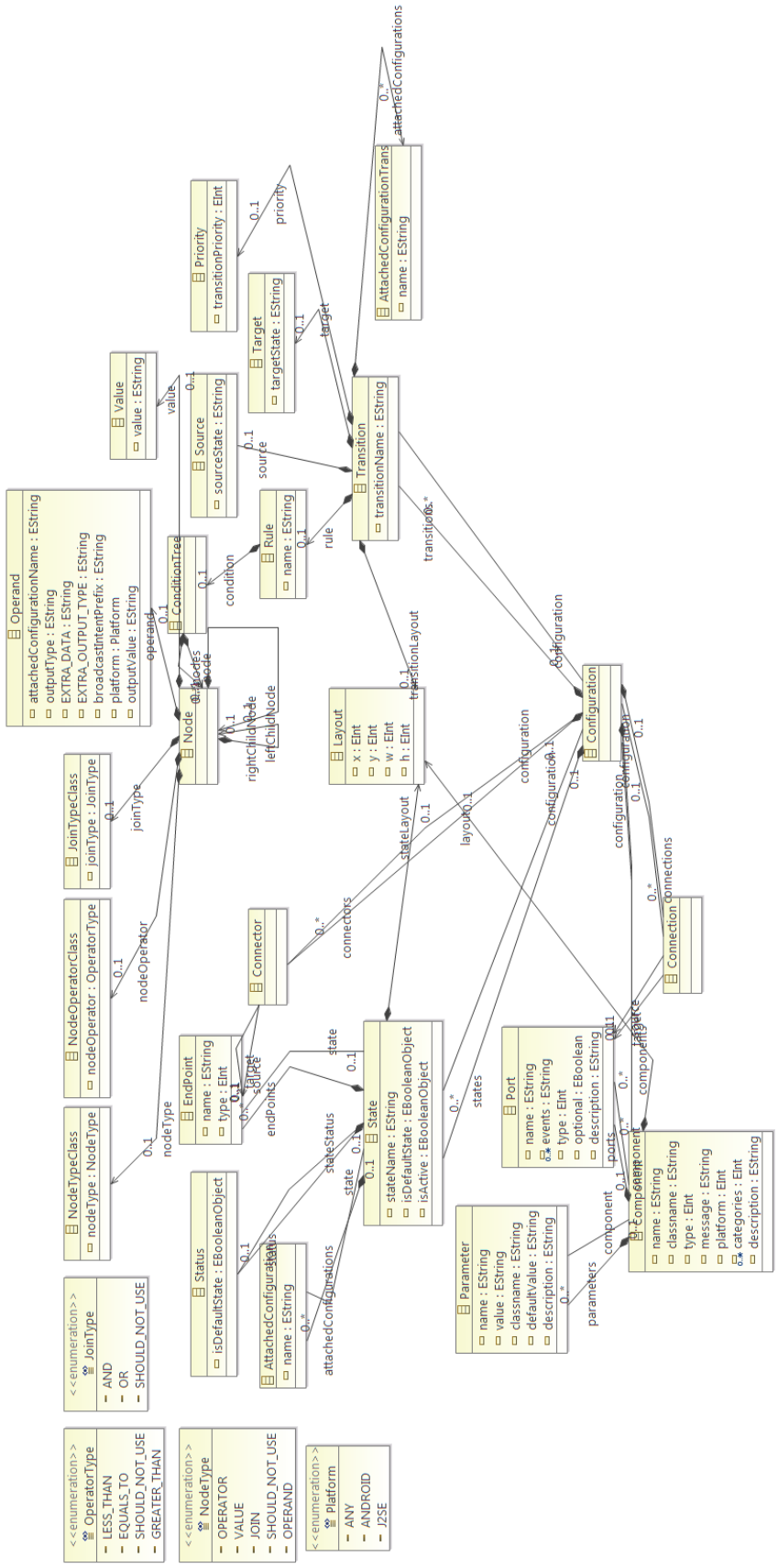

Figure 7.13: Eclipse modelling framework model of the activation system

7.4 Evaluation

The main design objective of activation system was to enable context dependent execution of only those configurations which are relevant to the current context of the user to enable energy savings for the resource constrained smart phones. Moreover, using the state machine abstraction the activation system enables applicability of four energy efficiency techniques namely Suppression, Substitution, Adaptation and Piggybacking for applications targeting any context type thereby supporting generic applicability of these techniques. Moreover, the graphical editor of the activation system aids in rapid creation of the required state machines. Therefore, the main attributes to evaluate for the activation system is the support to model applications such that they can utilize these energy efficiency techniques and the associated graphical editor. In this section, we demonstrate the generic applicability of these four techniques and the energy savings achieved using the activation system by creating a test application for performing indoor localization by using the graphical editor. The test application was realized using different configurations which are detailed in the next section. It is worth mentioning that we have chosen an indoor localization application primarily for illustrative purposes, since the various context recognition approaches are easy to explain and understand. The basic methodology and principles for integrating the four energy efficiency techniques in the form of certain state machine configurations, however, can be applied to any kind of application which supports the recognition of context via alternative approaches.

7.4.1 Indoor Localization Application

The indoor localization application has been built in two stages. In the first stage, we created the configurations required for performing the localization. In the second stage we created the state machines to arrange those configurations such that the Suppression, Substitution and Adaptation are used. As evaluation of configuration folding in the next chapter provides a detailed evaluation on Piggybacking, we skip its evaluation using the test application.

The configurations for localization were created using the graphical editor and the component toolkit provided by the component system. In total we created four configurations which include a configuration for measuring the battery status of an Android device (BatteryConfiguration) as basis for adaptation, a configuration to determine user movement using the device's accelerometer (AccelConfiguration), a configuration to determine user movement using the audio sensor (SoundConfiguration) and a configuration for performing the actual localization (LocalizationConfiguration).

In order to demonstrate Adaptation, we created a two level hierarchical state machine. At the higher level, one state machine was used to control the adaptation process. At the lower lever we created two state machines to perform the localization (using different strategies). These state machines were created using the graphical editor of the activation system. For highlighting the advantage of having the graphical editor and rapid prototyping it provides, the state machine at the higher level of the hierarchy is depicted in Figure 7.14

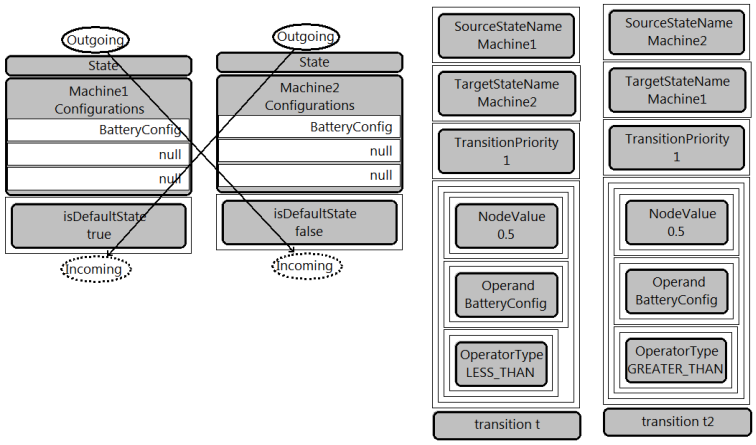

Figure 7.14: Screenshot of graphical editor showing the adaptation support

and its generated code snippet is depicted in Figure 7.15, Figure 7.16 and Figure 7.17. This state machine is used to switch between two state machines each representing different settings for the localization. To do this, this state machine used BatteryConfiguration in two states to start/stop one of the two state machines in the lower level of the hierarchy. The rule associated with the transition of the state machine at the higher level was set to use state machine 1 when the phone's battery is more than 50% and use state machine 2 when the phone's battery is less than 50%.

```
public StateMachineConfiguration createConfiguration(String configurationName) {
        StateMachineConfiguration configuration = new StateMachineConfiguration(configurationName);
        // create states configurations
        State state0 = new State();
        state0.setName("Machine1");
        state0.setDefaultState(true);
        BatteryConfig BatteryConfig0 = new BatteryConfig();
        state0.setConfigurations(BatteryConfig0.createConfiguration("BatteryConfig"));
        configuration.setState(state0);

        State state1 = new State();
        state1.setName("Machine2");
        state1.setDefaultState(false);
        BatteryConfig BatteryConfig1 = new BatteryConfig();
        state1.setConfigurations(BatteryConfig1.createConfiguration("BatteryConfig"));
        configuration.setState(state1);
```

Figure 7.15: Code snippet for state in the state machine generated by the graphical editor

In one of the two state machines, the LocalizationConfiguration was parametrized to perform continuous WiFi scans whereas for the other state machine the LocalizationCon-

figuration was parametrized to perform WiFi scans every 10 seconds. The time interval of 10 seconds is chosen to highlight the energy savings compared with continuous location recognition. To demonstrate Suppression, we added a second state to the state machines. This second state uses the AccelConfiguration in order to suppress the use of the LocalizationConfiguration in cases where the device is not moved (i.e. accelerated). In addition, we also added the AccelConfiguration to the state which had LocalizationConfiguration attached to it. To demonstrate Substitution, we copied the state machine and replaced the AccelConfiguration to determine the user movement with the SoundConfiguration. Though, accelerometer data is typically used for the determination of movement in scenarios where user is moving between places a distinct change of ambient noise can be an alternate source of movement information.

```
//creating transitions
Transition transition0 = new Transition();
transition0.setName("transition t");
transition0.setPriority(1);
transition0.setSourceState("Machine1");
transition0.setTargetState("Machine2");
Rule rule0 = new Rule();
rule0.setName("Rule");
ConditionTree conditionTree0 = new ConditionTree();
//setting up value
Value value0 = new Value();
value0.setValue("0.5");
//setting up operand
transition0.addConfigurationsAttacdhedToTransition("BatteryConfig");
Operand operand0 = new Operand(Platform.ANDROID,"BatteryConfig");
//setting up operator
Node node_operand0 = new Node(NodeType.OPERAND,null,null,null,operand0,null,null);
Node node_value0 = new Node(NodeType.VALUE,null,null,null,null,value0,null);
Node node_operator0= new
Node(NodeType.OPERATOR,node_operand0,node_value0,OperatorType.LESS_THAN,null,null,null);
//setting up nodes in the condition tree
conditionTree0.setCondArrayList(node_operand0);
conditionTree0.setCondArrayList(node_operator0);
conditionTree0.setCondArrayList(node_value0);
// setting up rule
rule0.setCondition(conditionTree0);// adding rule to transition
transition0.setRule(rule0);// adding transition to state machine
configuration.setTransition(transition0);
```

Figure 7.16: Code snippet for first transition generated by the graphical editor

```
//creating tranistion
Transition transition1 = new Transition();
transition1.setName("transition t2");
transition1.setPriority(1);
transition1.setSourceState("Machine2");
transition1.setTargetState("Machine1");
Rule rule1 = new Rule();
rule1.setName("Rule");
ConditionTree conditionTree1 = new ConditionTree();
//setting up value
Value value1 = new Value();
value1.setValue("0.5");
//setting up operand
transition1.addConfigurationsAttacdhedToTransition("BatteryConfig");
Operand operand1 = new Operand(Platform.ANDROID,"BatteryConfig");
//setting up operator
Node node_operand1 = new Node(NodeType.OPERAND,null,null,null,operand1,null,null);
Node node_value1 = new Node(NodeType.VALUE,null,null,null,null,value1,null);
Node node_operator1= new
Node(NodeType.OPERATOR,node_operand1,node_value1,OperatorType.GREATER_THAN,null,null,null);
//setting up nodes in the condition tree
conditionTree1.setCondArrayList(node_operand1);
conditionTree1.setCondArrayList(node_operator1);
conditionTree1.setCondArrayList(node_value1);
// setting up rule
rule1.setCondition(conditionTree1);// adding rule to transition
transition1.setRule(rule1);// adding transition to state machine
configuration.setTransition(transition1);
```

Figure 7.17: Code snippet for second transition generated by the graphical editor

7.4.2 Experiments

For our experiments with the application, we used a Samsung Galaxy Nexus phone running on Android 4.2 as our target platform. To determine the energy usage, we used a precise energy measurement hardware as shown in Figure 7.18 for measuring the power consumption. As shown in Figure 7.18, this means that we are connecting a high precision measurement resistor of 100 mΩ in series between the battery pack and the Samsung Galaxy and we connect both, the battery and the resistor to a high speed data logger (NI USB-6212) to measure the voltage of the battery as well as the power drain of the device. Since the sampling rate of the data logger is limited to 400kHz, we sample both inputs at 200kHz which is slightly lower than the 250kHz measurements performed in [RH10].

We first computed the base power drain with device's screen brightness set to minimum (0.609 watts). Thereafter, we performed additional measurements showcasing the three techniques. For Adaptation, the power consumption of the first state machine (performs continuous WiFi scans) in which the LocalizationConfiguration was parametrized to perform continuous WiFi scans was computed to be 0.96 Watts where as for the second

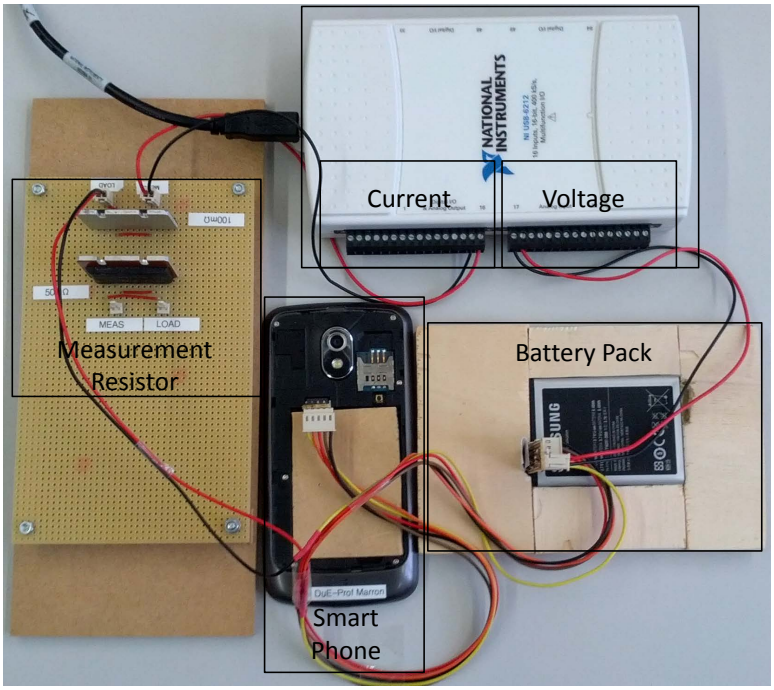

Figure 7.18: Power measurement setup

state machine in which the LocalizationConfiguration was parametrized to perform WiFi scans every 10 seconds was computed to be 0.634 Watts. In both cases the state machines were executed for 60 seconds. We subtracted base power consumption to obtain actual power consumptions. For first state machine the actual power consumption was calculated to be 0.96-0.609 = 0.351 Watts where as for the second state machine it was 0.634-0.609 = 0.0252 Watts. The net power saving for executing the localization with different setting were computed to be (1-0.0252/0.351)*100 = 92%. The power consumption graphs for Adaptation are shown in Figure 7.19. In this figure, the state machine 1 refers to the first state machine which performs localization continuously where as state machine 2 refers to the second state machine which performs localization with time interval of 10 seconds. The graph for the state machine 1 shows continuous high power measurements which can be attributed to continuous processing of CPU at a higher or maximum capacity where as the graph of state machine 2 shows periodic pulses at lower wattage and with an approximate intervals of 10 seconds. The graph labelled as base in this figure refers to the measurements when activation system is not running and the brightness of the device screen is set to minimum.

For Suppression, we assumed that user does not move for half of the execution time of the application i.e. for 30 seconds out of 60 seconds (similar to experiment settings in [ZKS10]), the user remains static. As a result, the state machine used for Suppression remains in state attached with AccelConfiguration for 30 seconds and switches to other state attached with AccelActivation and LocalizationConfiguration both for the remaining 30 seconds. The average power consumption for this state machine was computed to be

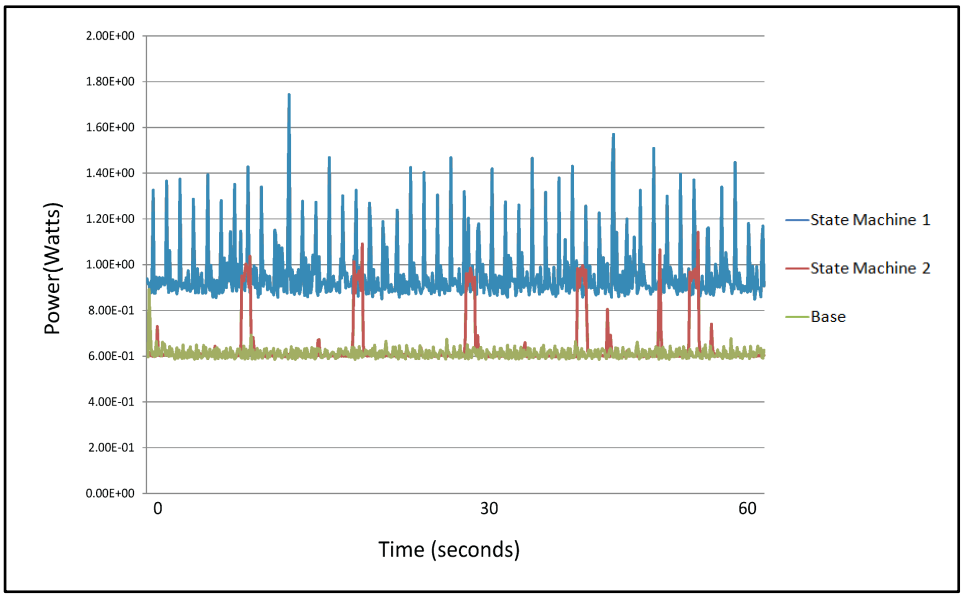

Figure 7.19: Power consumption when Adaptation is used.

0.842 Watts. After subtracting the base power the actual power consumption becomes 0.842-0.609 = 0.233 Watts. In order to measure the power savings for Suppression, we computed power consumption of state machine continuously executing LocalizationConfiguration. Its power consumption was computed to be 0.946-0.609 = 0.337 Watts. Hence, the net power savings for Suppression were computed to be (1-0.233/0.337)*100 = 30.8%. The power consumption graph for Suppression is shown in Figure 7.20. We can see that the power consumption of state machine 1 is almost same throughout the time since it is continuously performing localization. The power consumption of state machine 2 remains low for about period of 30 seconds. This is due to the fact that user is not moving. Afterwards, as user starts to move, the localization is performed and hence the power consumption is increased. It can be seen in the figure that during this time period the power consumption of state machine 2 is higher than state machine 1. This is due to the fact that during this period state 2 of state machine 2 is executing both, the LocalizationConfiguration and the AccelConfiguration.

For Substitution, the actual power consumption for state machine with SoundConfiguration was computed to be 0.982-0.609 = 0.373 Watts and for state machine with AccelConfiguration was computed to be 0.675-.609 = 0.0662 Watts. The net power savings if AccelConfiguration is used instead of SoundConfiguration were (1-0.0662/0.373)*100 = 82.2%. The power consumption graph for Substitution is shown in Figure 7.21. This figure illustrates comparison of determining user movement with two different methods as per the definition of Substitution. The graph for state machine 1 shows the power consumption of running SoundConfiguration to determine the user movement where as graph for state machine 2 shows the power consumption of running the AccelConfiguraiton for determining the user movement.

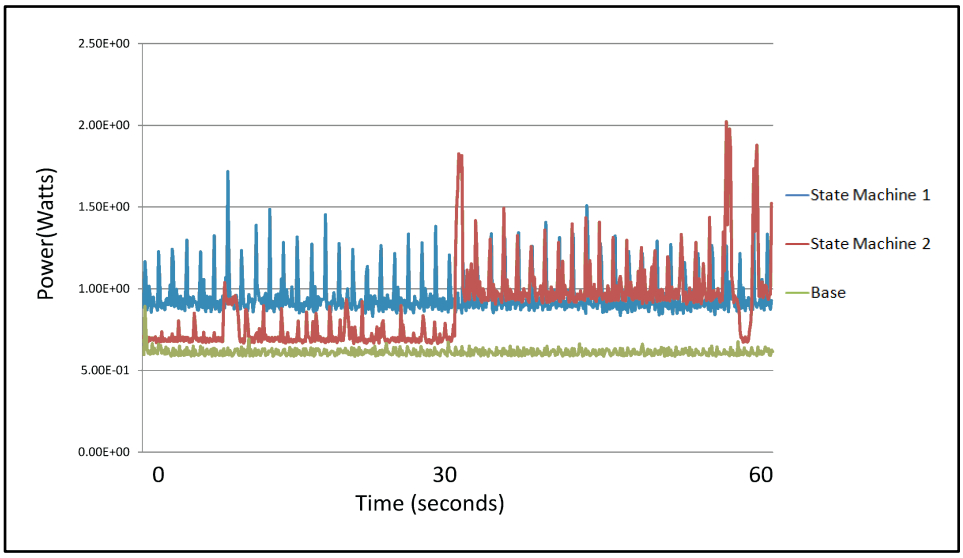

Figure 7.20: Power consumption when Suppression is used.

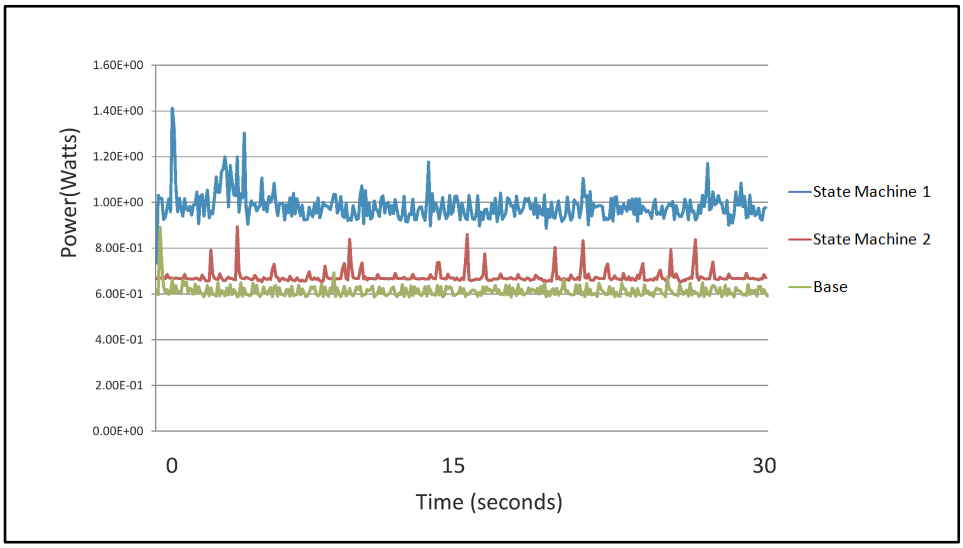

Figure 7.21: Power consumption when Substitution is used.

These energy savings demonstrate that using state machine abstraction is a viable approach to achieve context dependent activation of relevant configurations or applications in general and for enabling generic applicability of Suppression, Substitution, Adaptation and Piggybacking in particular. The above measurements does not show experimental evaluation for Piggybacking because as we have indicated earlier in the chapter that the activation system uses configuration folding to achieve Piggybacking. A thorough evaluation of Piggybacking is performed in the chapter on configuration folding.

7.4.3 Fitness Monitoring Application

The fitness monitoring application has been created to demonstrate that the state machine abstraction can be applied to any application targeting any context type and not just the localization. This application recognizes different user movement modalities such as standing, jogging and running. The application has been created using the graphical editor of the activation system as shown in Figure 7.22. The application consists of three states. A configuration AccelActivation for determining user's movement is attached to each state. Different rules are assigned for transitions between the states. When the mean acceleration is greater than 10 then the user is assumed to be jogging and when it is greater than 14 then user is assumed to be running. Similarly, if the mean acceleration is less than 14 then user is assumed to be jogging and when it is less than 10, the user is assumed to be standing.

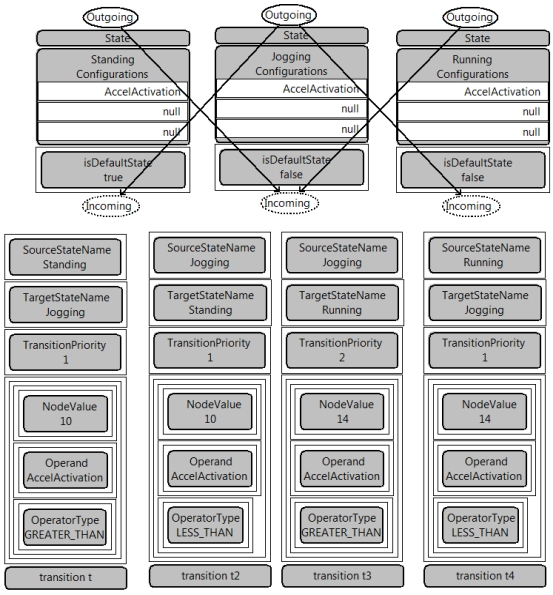

Figure 7.22: State machine configuration used in the fitness monitoring application

The above mentioned localization and fitness monitoring applications developed by using our activation system demonstrate that the system is capable of supporting creation of applications targeting different context. Moreover, the state machine abstraction can be

used to model applications such that the four energy efficiency techniques can be applied irrespective of their target context type. The configuration and state machines for these applications has been created using the graphical editors of the component system and the activation system respectively which highlight the advantages of rapid prototyping tools. For configurations and state machines, the code consist of repetitive constructs of component and state machine abstractions of considerable length, writing it manually is prone to errors and also time consuming.

7.4.4 Requirement Coverage

One of the four design requirements for our framework is Efficiency. The requirement on efficiency is described at two levels namely the efficiency for creating the context recognition applications and efficiency for executing these applications in an energy efficiency manner. The activation system supports both. If a context recognition application uses more than one configuration to recognize the context and if every configuration is not required at all the time, then the state machine abstraction used by the activation system enables execution of only relevant context recognition configurations such that the management of starting and stopping the configurations is not part of the application logic. Thereby, activation system provides a seamless integration of this energy efficient execution with the applications. In addition, the state machine abstraction also enables generic applicability of four energy efficiency techniques. This is demonstrated in the localization application above. It is evident from the description of this example that the same state machine model can be used for any other context e.g. fitness monitoring, activity recognition etc. and not just the localization. Moreover, the tools support of activation system enables rapid prototyping of such applications by allowing developers to create visual representation of the state machines and generate their code and directly use it in the application. Thus, we can say that activation system adds support for efficiency to our framework and thereby fulfils the design requirement on efficiency.

7.5 Summary

This chapter has provided an in-depth detail about the activation system. It discussed the state machine abstractions used by the activation system which consists of states and transitions. It also discussed the internal working of the activation system. In the chapter we also showed that by using the activation system it is possible to provide generic applicability of four energy efficiency techniques which have been used for specific contexts in the existing work. We discussed the off-line development tools of the activation system. We discussed various aspects of the graphical editor and showed the rapid prototyping support it provides. We also evaluated the activation system by creating a test application using its graphical editor and by performing precise energy measurements on the application to demonstrate the energy savings provided by the activation system.

8 CONFIGURATION FOLDING

In this chapter we discuss our novel energy efficiency technique configuration folding. The configuration folding allows energy efficient execution of simultaneously executing applications. In this chapter we formalize the problem that configuration folding solves and then first presents the basic configuration folding algorithm. We also discuss how configuration folding can be applied if the simultaneously executing applications differ in parametrization. We investigate different commonly used parameters in context recognition applications and identify relations between them. We then present the extended folding algorithm which also takes parametrization differenced into account. Finally, we conclude the chapter by evaluating configuration folding using test applications and performing precise energy measurements on them.

8.1 Configuration Folding

Configuration folding builds upon existing context recognition applications in that it aims at minimizing their energy requirements. However, instead of focusing on one concrete application, configuration folding targets the simultaneous execution of multiple. Consequently, it is related to the energy efficiency provided by the frameworks such as [NKL+07] or [KLJ+08]. Yet, instead of focusing on specific types of context information that are known at system design time, configuration folding is open and generic in the sense that it can be applied on any number of context recognition applications. Specifically, configuration folding solves energy consumption in simultaneously executing applications by identifying and removing redundancies between them. This requires analysis of application structure at runtime since a user might be starting and stopping different applications in different point in time. In order to realize the analysis of application's structure, we rely on the component abstraction employed by our framework. As context recognition applications developed using our framework realize their recognition logic through configurations (set of components and connectors), it enables component system to know about the structure of applications by analysing these configurations.

8.1.1 Problem Analysis

Given the component model described in the chapter on component system, we can informally define redundancy as follows. Given two configurations, a set of components is redundant if the set is contained in both configurations in such a way that all identical

components exhibit the same connections and parametrizations and that each set does not depend on the inputs of components that are not part of the set. Based on this definition, the outputs generated by both sets of components cannot differ. To justify this consider that components at the sampling level do not require inputs from other components. Consequently, if two configurations contain the same sampling component with an identical parametrization, it is possible to remove one of the two components by reconnecting the components that require its output to the other component. If the components require inputs, however, it is also necessary to ensure that the inputs are identical in order to ensure that their outputs cannot differ.

Redundancy

In order to formalize the redundancy between configurations, we can think of a configuration as a directed acyclic graph (DAG) in which the vertices represent the components and the edges represent the data- and control-flow established by connectors. To model the configuration details, we introduce a function $F(vertex) \rightarrow (impl, params)$ that maps each vertex to its associated implementation and parametrization. Given two configurations $G_1 = \{V_1, E_1\}$ with F_1, $G_2 = \{V_2, E_2\}$ with F_2, we can define redundancy as a subset S of V_1 and an associated bijective mapping $R : s \rightarrow v$ with $s \in S, v \in V_2$ subject to the following conditions. First, we ensure that the redundant wiring is identical, i.e. has the same edges.

$$\forall s_1, s_2 \in S : (s_1, s_2) \in E_1 \leftrightarrow (R(s_1), R(s_2)) \in E_2 \tag{8.1}$$

Second, we ensure that redundant vertices represent the same components with the same parametrization.

$$\forall s \in S : F_1(s) = F_2(R(s)) \tag{8.2}$$

Finally, we ensure that neither components in V_1 nor components in V_2 require non-redundant inputs.

$$\forall s \in S, v \in V_1 : (v, s) \in E_1 \rightarrow v \in S \tag{8.3}$$

$$\forall s \in S, v \in V_2 : (v, R(s)) \in E_2 \rightarrow R^{-1}(v) \in S \tag{8.4}$$

Given the condition (8.1) alone, the problem would correspond to finding a homomorphism in a sub graph of G1 and G2 which is known to be NP complete. However, due to equations (8.2), (8.3) and (8.4) as well as the DAG structure of configurations, it is possible to find an efficient solution.

8.1.2 Basic Algorithm

Our configuration folding algorithm is based on the idea that two components can only be redundant if they are on the same level in the topological ordering of the two graphs. If they are not on the same level, the component at the higher level will be connected to at least one component that was not present in the other configuration. Consequently, in each step of the algorithm, we can restrict comparisons between the graphs to the set of vertices at

a particular level. By traversing the graph in topological order, we can then constructively ensure the conditions (8.1) and (8.2) by matching the component descriptions and we can ensure condition (8.3) and (8.4) recursively by ensuring that all incoming edges are originating in a redundant component. To do this efficiently, we incrementally replace the incoming edges in the input graphs with labels to their parent vertices in the folded graph. The resulting algorithm is shown in Listing 8.1.

```
1  Define Graph As            // configuration
2    Vertices: Set<Vertex> // all components
3  Define Vertex As           // component & connectors
4    Incoming: Set<Vertex> // connected inputs
5    Outgoing: Set<Vertex> // connected outputs
6    Labels  : Set<Vertex> // labels of inputs
7    Comp    : Component   // component impl.
8
9  FoldGraphs(Graph G1, Graph G2): Graph
10   Graph Res = New Graph
11   // start on first level
12   Set<Vertex> Cur1 = GetRoots(G1)
13   Set<Vertex> Cur2 = GetRoots(G2)
14   While (! (Cur1.IsEmpty() & Cur2.IsEmpty()))
15     Set<Vertex> Next1 = New Set
16     Set<Vertex> Next2 = New Set
17     // create hash map for constant lookup
18     Map<Component, Vertex> M = New Map
19     Foreach (Vertex N2 In Cur2)
20       M.Put(N2.Comp, N2)
21     // handle vertices that can be folded
22     Foreach (Vertex N1 In Cur1)
23       Node N2 = M.Get(N1.Comp)
24       If (N2 != NULL && AllowsFolding(N1,N2))
25         Vertex N = AddVertex(Res, N1.Comp)
26         Visit(N, N1, Next1)
27         Visit(N, N2, Next2)
28         Cur1.Remove(N1)
29         Cur2.Remove(N2)
30     // handle vertices that cannot be folded
31     Foreach (Vertex N1 In Cur1)
32       Vertex N = AddVertex(Res, N1.Comp)
33       Visit(N, N1, Next1)
34     Foreach (Vertex N2 In Cur2)
35       Vertex N = AddVertex(Res, N2.Comp)
36       Visit(N, N2, Next2)
37     // continue with next level
38     Cur1 = Next1
39     Cur2 = Next2
```

```
40   Return Res
41
42 Visit(Vertex Nu, Vertex Ol, Set<Vertex> S)
43   // connect according to configuration
44   Foreach (Vertex N In Ol.Labels)
45     Nu.Incoming.Add(N)
46     N.Outgoing.Add(Nu)
47   // label edges and compute next level
48   Foreach (Vertex N In Ol.Outgoing)
49     N.Incoming.Remove(Ol) // mark parent done
50     N.Labels.Add(Nu)    // memorize connection
51     If (N.Incoming.isEmpty())
52       S.Add(N)    // no inc. -> parents done
53
54 AllowsFolding(Vertex N1, Vertex N2): Boolean
55   Return N1.Comp.Equals(N2.Comp) &
56          N1.Labels.Equals(N2.Labels)
57
58 AddVertex(Graph G, Component C): Vertex
59   Vertex Res = New Vertex[Comp=C]
60   G.Vertices.Add(Res)
61   Return Res
62
63 GetRoots(Graph G): Set<Vertex>
64   Set<Vertex> Res = new Set
65   Foreach (Vertex N In G.Vertices)
66     If (N.Incoming.IsEmpty()) Res.Add(N)
67   Return Res
```

Listing 8.1: Basic Configuration Folding Algorithm

As depicted in Listing 8.1, the folding algorithm, i.e., *FoldGraphs*, starts by determining the lowest level in each graph using *GetRoots*. Then it checks whether two vertices on that level are redundant. To allow constant time comparisons, the vertices of one of the graphs is hashed by components (for simplicity, we omit collision handling). When two vertices represent identical components, *AllowsFolding* additionally checks whether their incoming edges have been labeled identically, i.e. whether they originate in the same set of components – which enforces condition (8.2). On the lowest level of the graph, this will always be the case since the roots have no inputs. For consecutive levels, this will only be the case for redundant components – which enforces the conditions (8.3) and (8.4) recursively. If redundant components have been identified, a new vertex is created using *AddVertex*. Then, the new vertex is connected according to the configuration using *Visit*. Thereby, the visit method will add labels to all vertices that are connected to the vertex that is about to be folded. Furthermore, to speed up the traversal, the *Visit* also determines the vertices on the next level of the graph. This is done by removing incoming edges in the outgoing vertices and testing for emptiness. Note that this simply reflects

topological traversal as described in [Kah62]. Once all possible vertices have been folded, the remaining vertices are handled by creating new vertices and adjusting the connections while computing the vertices on the next level in *Visit*. Thereafter, the algorithm switches to the next level in both graphs, which has been gathered already. After handling all levels, the algorithm returns the folded graph.

To analyse the algorithm's complexity, one must consider that the algorithm simply traverses both graphs in topological order. Thereby, each vertex and each edge is visited exactly once. Due to the fact that the comparisons between the vertices on each level are performed in constant time using hashing, the overall complexity of topological sorting is not increased. Consequently, the complexity of the algorithm is $O(n_1 + m_1 + n_2 + m_2)$ where n_i and m_i represent the number of vertices and edges in the graphs which results in $O(n_1^2 + n_2^2)$ since there are at most n_i^2 edges.

Piggybacking Support

We mentioned in the chapter on the activation system that the activation system achieves Piggybacking through configuration folding. As Piggybacking refers to multiplexing of resources between different applications, configuration folding also achieves the same by exploiting redundancy between different applications and by using a single folded configuration to multiplex the functionalities. The Piggybacking defined by the authors in [ZKS10] was specific to location sensing applications whereas configuration folding can be applied to any context types. Due to this support provided by configuration folding, there is no need to have a separate implementation of Piggybacking in the activation system. In the activation system the Piggybacking is applied using the component system as soon as the state machine enters in a state which has more than one configuration attached to it.

8.2 Parametrization

Configuration folding allows energy efficient execution of simultaneously executing multiple application and it does so by identifying and removing redundancies between the components in the input configurations. However, the energy savings achieved using configuration folding could be limited if the redundant components differ in parametrization. In order to understand the effect of parametrization consider the example shown in Figure 8.1. The Figure 8.1(a) shows a configuration of honk detection application and Figure 8.1(b) shows configuration for music detection application. Since both of these configurations use same microphone component sampling at 8kHz and a Fast Fourier Transform component and these component fulfil the conditions of redundancy, their folded configuration is shown in Figure 8.1(c). Now consider a case where honk detection application uses the microphone component with 16kHz and music detection application uses microphone component with 8kHz. According to the conditions of redundancy given above, this would violate condition (8.2) and the folding of these two components would not be possible, thereby the folding process would stop and consequently loose potential energy savings.

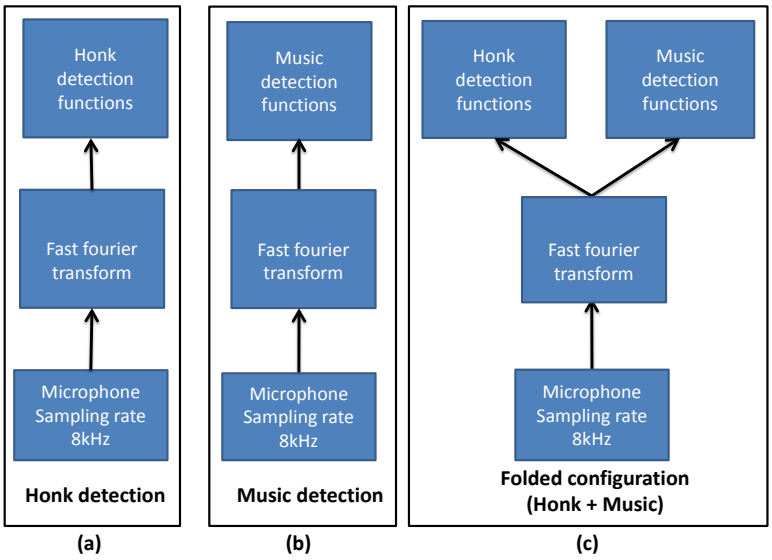

Figure 8.1: Configuration folding example

In order to avoid such situation and increase the possible savings of configuration folding due to differences in parametrization we rely on transformation components described in the chapter on component system. We use transformation components in the the folded configurations to avoid parametrization differences. Using transformation components allow us to rewrite a folded configuration such that instead of exhibiting differences in parameters, it exhibits additional components. Figure 8.2 illustrates an example of use of transformation component. Figure 8.2(a) shows that the configuration for honk detection application uses audio sampling rate of 16kHz where as Figure 8.2(b) shows the configuration for music detection application uses audio sampling rate of 8kHz. The folded configuration shown in Figure 8.2(c) uses the audio component with sampling rate of 16kHz. In addition, the folded configuration also uses a transformation component which takes input from the 16kHz microphone component and transforms it to the equivalent output of 8kHz microphone component. This example also highlights that use of transformation component is possible only if certain relation between the parameter values exists such that the conditions of redundancy are not violated. As we can see that one microphone component has a sampling rate of 16kHz and the other 8kHz meaning there exist a two to one relation between the parameter values between the components. The transformation component in this example forwards every second value from the 16kHz component which would be equivalent to the output of 8kHz component, the opposite of which is not possible.

8.2.1 Handling Common Parameters

In this section, we present commonly used parameters in the context recognition applications. These parameters have been identified by analysing a number of context recognition applications that we have developed using the component system over the last 4 years.

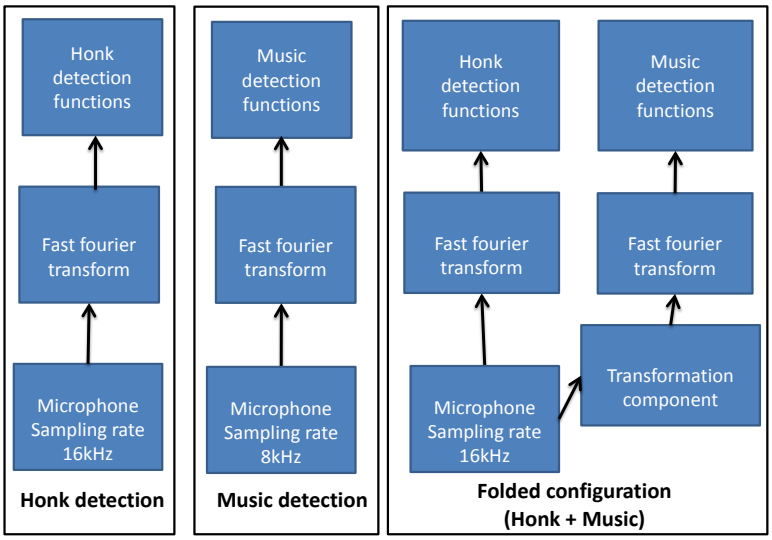

Figure 8.2: Configuration folding with transformation component

Among others, they include a music and speech recognition application [IHW+12b], collaborative GPS sharing application [AHIM14], crowd-level estimation application [HIW+14], speaker recognition application [AHIM13] and assisted living and mobility applications that have been developed in the LIVING++ [IFW+13] and in the GAMBAS [gam] [AIP14] European research projects.

- Sampling Rate: It is defined as the number of data samples sampled per second. This parameter type is mostly used at the sensing level of a configuration. The possible components using this type are audio sensing components, acceleration sensing components, location sensing components etc. Based on our experiences with the developed components, the sampling rate has a consensual meaning i.e. component developers have the same understanding of sampling rate.

- Sampling Depth: It is defined as the number of bits representing a sample value. This parameter type is mostly used at the sensing level of a configuration. The possible components for this type are all the sensor sampling components. Based on our experience, this parameter type has a consensual meaning among developers.

- Threshold: It is defined as a border line value that is used for making a decision. This parameter type is mostly used at the preprocessing and classification level of a configuration. The possible components using this type of parameter includes high pass and low pass filters, time-frequency domain transformation etc. Based on our experience, this parameter type does not have a consensual understanding among component developers and needs further classification according to the component type e.g. high frequency threshold for high pass filter component, Fast Fourier Transform threshold for the FFT component etc.

- Window Size: It is defined as the size of data values. This parameter type is mostly used at the sensing and preprocessing level of a configuration. This parameter type does not have consensual understanding at preprocessing level components but for the sensing level components it has consensual understanding. At the sensing level it represents data size of the sampled data output by the component.

In the following we discuss the possible relations between the identified parameter types which have consensual understanding of their meanings. The use of transformation component for the components using above mentioned parameter types is possible only if the following relations exist between their values.

Sampling Rate Relations

Let $C_{xij}^{P_a}$ and $C_{yik}^{P_b}$ be two components such that x equals y and P_a and P_b represent sampling rate type parameter. The possible relations for this parameter type are

- 1 to N where N $>$ 1 : If $C_{xij}^{P_a}$ samples one sample every second and $C_{yik}^{P_b}$ samples N samples every seconds then the transformation component ($TC_z^{P_{ab}}$) will take the output of $C_{yik}^{P_b}$ at its input and outputs every Nth input data value.

- M to N : If $C_{xij}^{P_a}$ samples at M samples per second and $C_{yik}^{P_b}$ samples at N samples per seconds such that M/N is an integer value than the transformation component will take output of $C_{xij}^{P_a}$ at its input and outputs every Mth/Nth input data. Similarly if N/M is an integer value then $TC_z^{P_{ab}}$ will take the output of $C_{yik}^{P_b}$ at its input and outputs every Nth/Mth input data value.

Sampling Depth Relation

Let $C_{xij}^{P_a}$ and $C_{yik}^{P_b}$ be two components such that x equals y and P_a and P_b represent sampling depth type parameter. The possible relations for this parameter type are

- M $>$ N : If the sampling depth of samples sampled by $C_{xij}^{P_a}$ is M and the sampling depth of samples by $C_{yik}^{P_b}$ is N such that M $>$ N then $TC_z^{P_{ab}}$ will take output of $C_{xij}^{P_a}$ at its input and outputs the input data. Similarly if N $>$ M then $TC_z^{P_{ab}}$ will take the output of $C_{yik}^{P_b}$ at its input and outputs input data value. This parameter type is linked with sampling rate parameter type and to use a transformation component it is necessary that chosen component is same for the both parameter types.

Window Size Relation

Like sampling depth, this parameter type is also related with sampling rate of a sampling component. The possible relation for this parameter type is

- M to N: Let $C_{xij}^{P_a}$ and $C_{yik}^{P_b}$ be two components such that x equals y and P_a and P_b represent window size type parameter. Let's assume that according to the relation of sampling rate parameter type discussed previously, $C_{xij}^{P_a}$ is chosen for the folded configuration and $TC_z^{P_{ab}}$ outputs data equivalent to the data of $C_{yik}^{P_b}$. The use of transformation component in this setting only holds true if the value of P_a = (sampling rate of $C_{xij}^{P_a}$/ sampling rate of $C_{yik}^{P_b}$) $* P_b$.

These mentioned parameter type and their possible relations for the use of transformation component is based on our experiences with the development of a number of context recognition applications. This list of parameters might grow in the future. However, for new parameter types it might be necessary to have more fine grained type categories such as "$FilterThreshold$" instead of just "$Threshold$". The relations discussed for the parameters types ensure that the transformation components produce exactly the same output as would have been produced by the component they replace. Thereby use of transformation component in the folded configuration does not affect the correctness of the input configurations and the output produced by the folding component is exactly the same as would have been produced by input configuration individually. However some existing work such as [JLT04] also suggest that changing the sampling rate and sensor signal resolution with in some range does not have significant impact on overall classification accuracy. This could mean that if we relax the relations between the parameter values, configuration folding can yield even better energy savings.

8.2.2 Handling Data Types

The data types of input and output ports of a transformation component plays a crucial role in deciding whether the transformation component can be used at all in the folded configuration. These data types need to be consistent with the component they replace. Inconsistency in data type would either prohibit the configuration folding or would require some additional components for assuring data type consistency. There are six possible cases in which the transformation component is connected to other components. These cases are depicted in Figure 8.3. The first two cases (case 1 and case 2) deals with the situation when the component in the folded configuration sends its output to the input port of the transformation component. The first case deals with the situation when the component sending the output has same output data type as the input port of the transformation component. This situation would not require any additional data type adaptation functionality. The second case deals with the situation when the component sending the output has different data type than the transformation component. This situation would require additional data type adaptation functionality. The second two cases (case 3 and case 4) deals with the situation when the transformation component sends its output to the input port of other component. Case 3 deals with the situation when the output port of the transformation component and the input port of the other component have same data types. In this case no additional data type adaptation would be required. Case 4 deals with the situation when the data types are not same and would require additional

data type adaptation. The last two cases (case 5 and case 6) deal with the situation when one transformation component sends output to other transformation component. Like in previous cases if the output data type of one transformation component is not same as the input data type of other then this would require additional data type adaptation functionality, otherwise not. An example of data type consistency functionality is a component which takes input of "short" data type and outputs "double" data type. The extended configuration algorithm described in this chapter assumes that the input and output ports of transformation components have data types same as the components they replace. For delaying the use of transformation component in the folded configuration described in the next section, the algorithm assumes that the data type of the input and output ports of the components which allow delay of use of transformation component is same as that of the transformation component.

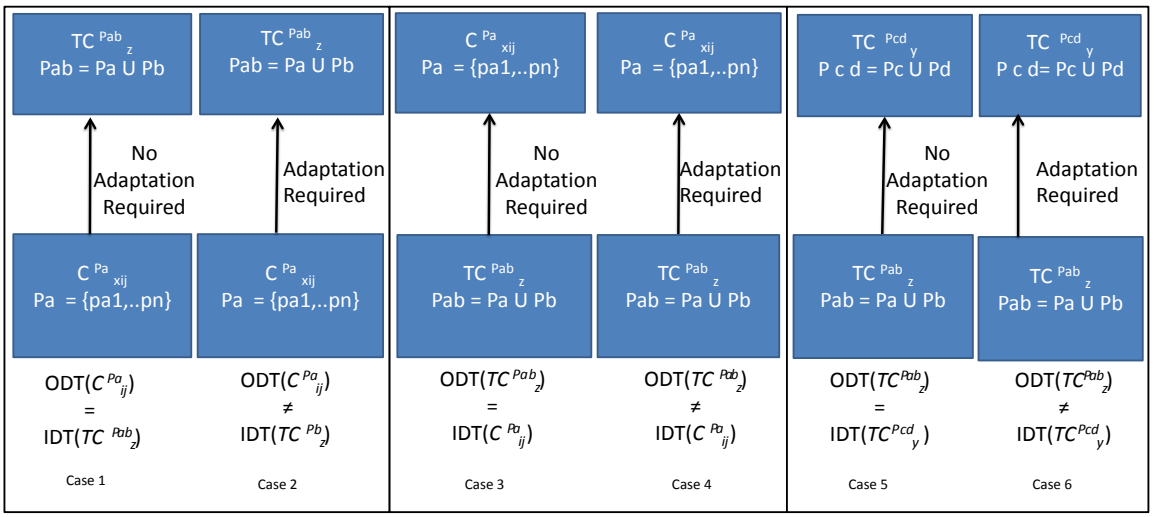

Figure 8.3: Cases for data type consistency in transformation component

8.2.3 Delaying Transformation Components

As described in the definition of redundant components, the conditions of redundancy states that configuration folding is possible if the underlying graph structures for the two components are same. Though use of transformation component overcomes the difference in parametrization but it does not stop the violation of the redundancy conditions. This is illustrated in Figure 8.2. The honk detection configuration in Figure 8.2(a) and music detection configuration in Figure 8.2(b) shows that both configuration have a FFT component but since these components expects different inputs, one from microphone component sampling at 16kHz and other sampling at 8KHz the FFT components cannot be folded. The use of transformation component for transforming the output of microphone component sampling at 16kHz to the output of microphone component sampling at 8kHz saves energy consumption by not using one microphone sampling component but it does not

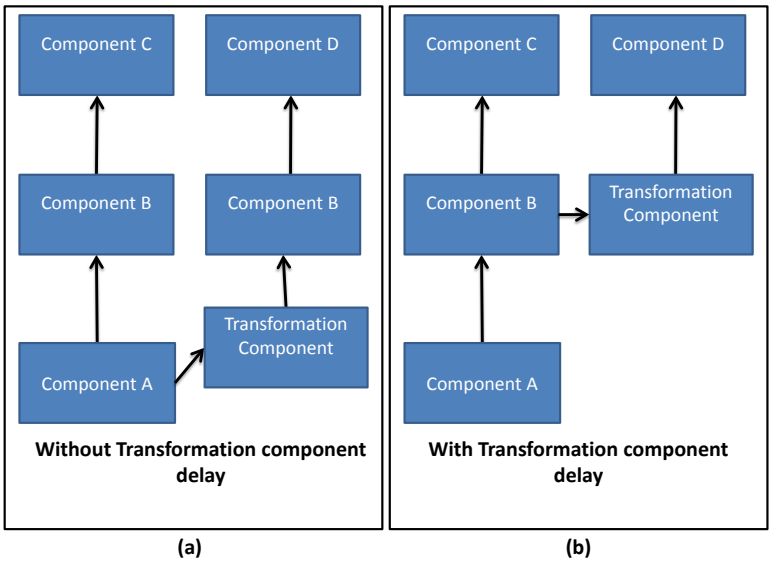

Figure 8.4: Configuration folding with transformation component delay

make the remaining components in the two configurations redundant. This means that as soon a transformation component is used, further configuration folding is not possible at the higher levels of the configurations and the folding algorithm stops.

In order to enable folding when transformation component has been used, we investigate the possibilities of delaying the use of transformation component in the folded configuration such that folding at higher levels of configurations is also possible. Figure 8.4 illustrates this goal. Figure 8.4(a) shows a folded configuration where a transformation component has been used to replace component A of one configuration due to difference in parametrization. Figure 8.4(b) shows that the use of this transformation component is delayed allowing component B to be folded as well. This delaying of transformation component can be achieved in two ways, either at the component level or at the configuration level.

Component Level

At the component level, this can be achieved when the components at one level above the transformation component possesses following two properties. For example the component B in Figure 8.4(b) must have these properties in order to allow delay of transformation component.

Property 1: The input and output data type of component is same as the input data type of the transformation component it delays

$$IDT(C_{xij}^{P_a}) = ODT(C_{xij}^{P_a}) = IDT(TC_z^{P_{ab}}) \tag{8.5}$$

Property 1 is necessary for the components to have input-output data type consistency between the output of the folded component and the input of the transformation component. The delaying of transformation component in Figure 8.4(b) is possible as long as output data type of component B is same as the input data type of the transformation component.

Property 2: The component is stateless (i.e., each output value of component is only a function of input value)

If $I(C_{xij}^{P_a}) = \{I_1, I_2, .., I_n\}$ and $O(C_{xij}^{P_a}) = \{O_1, O_2, .., O_n\}$ describes the set of input and output values of a component such that the cardinality of input $|I(C_{xij}^{P_a})|$ and output $|O(C_{xij}^{P_a})|$ is equal then the component is said to hold property 2 if it fulfils the following condition.

$$O_t = f(I_t) \tag{8.6}$$

where t is the index of output value and ranges from 1 to n where n is the cardinality of the input and output. Based on our analysis of the existing applications and our component toolkit, components that exhibit this behavior often fall into the following two categories:

- Arithmetic components: These components perform different arithmetic functions such as addition, subtraction, multiplication and division on the received input value(s). The output value(s) are computed by performing the arithmetic function on input value(s) with some specified operand e.g. if it is an addition arithmetic component then output values are computed by adding an operand to the input values.

- Marking components: These components add markers to the individual values of input data e.g. labelling an input value with some context characteristic such as "zero crossing rate" or "force" in order to compose feature vectors.

However, in addition to these general classes, there are several individual components in our toolkit that also exhibit these properties. An example is a stateless classifier that classifies its input values into certain value ranges.

Configuration Level

If a component exhibits the two properties described previously, it is safe to delay transformation components since the delaying is guaranteed to produce the same result. However, even if Property 2 does not hold for a particular component, it might still be safe (for a particular application) to delay the transformation to a later state. To enable this, we allow the application developer to mark the components that support the delaying of a transformation ((e.g Component B in Figure 8.4(b)).

As a result, the correctness of the resulting folded configuration is not guaranteed by the component system but is decided by the configuration developer. Having this flexibility at the configuration level to allow delay of transformation components for such components may provide energy savings for cases where configuration developer knows that doing so would not break the correctness of input configurations. In case the marked component can also be folded and requires a transformation component, then the provisioning of this transformation component is the responsibility of the configuration developer.

8.2.4 Extended Algorithm

In this section we provide the extended configuration folding algorithm with support for transformation components to handle parametrization and support for allowing delay of use of transformation component in a folded configuration. This algorithm is an extended version of the basic algorithm described earlier.

```
1  Definition:
2   Conf_1: Input configuration 1
3   Conf_2: Input configuration 2
4   Conf_R: Folded configuration
5   C^{P_a}_{xiConf_1}: Component in Conf_1
6   C^{P_b}_{yiConf_2}: Component in Conf_2
7   C^{P_c}_{chosenComp_12}: Chosen component
8   TC: Acronym for Transformation component
9   List< TC^{P_ab}_z > TC_LIST: List of TC
10 Working:
11 for all C^{P_a}_{xiConf_1} in Conf_1 at level i
12   for all C^{P_b}_{yiConf_2} in Conf_2 at level i
13     case = checkFoldingCase(C^{P_a}_{xiConf_1}, C^{P_b}_{yiConf_2})
14     If case = Case1
15       C^{P_c}_{chosenComp_12} =GetFoldedComp(C^{P_a}_{xiConf_1}, C^{P_b}_{yiConf_2})
16       Add (C^{P_c}_{chosenComp_12}, Conf_R)
17       UpdateLabelsAndConnect()
18     If case = Case2
19       C^{P_c}_{chosenComp_12} =GetFoldedComp(C^{P_a}_{xiConf_1}, C^{P_b}_{yiConf_2})
20       If C^{P_c}_{chosenComp_12} has TC
```

```
21      Add (C^{P_c}_{chosenComp_12}, Conf_R)
22        If true = DelayProperty(C^{P_a}_{xiConf_1}, C^{P_b}_{yiConf_2})
23          TC^{P_ab}_z = GetTC(C^{P_c}_{chosenComp_12})
24          Add TC^{P_ab}_z in TC_LIST
25          UpdateLabelsAndConnect()
26        Else
27          Add (TC^{P_ab}_z, Conf_R)
28          UpdateLabelsAndConnect()
29      Else goto Stop
30    If case = Case3
31      C^{P_c}_{chosenComp_12} =GetFoldedComp(C^{P_a}_{xiConf_1}, C^{P_b}_{yiConf_2})
32      Add (C^{P_c}_{choseniConf_12}, Conf_R)
33        If true = DelayProperty(C^{P_a}_{xiConf_1}, C^{P_b}_{yiConf_2})
34          UpdateLabelsAndConnect()
35        Else
36          for all TC in TC_LIST
37            Add (TC, Conf_R)
38            UpdateLabelsAndConnect()
39    If case = Case4
40      C^{P_c}_{chosenComp_12} =GetFoldedComp(C^{P_a}_{xiConf_1}, C^{P_b}_{yiConf_2})
41      If C^{P_c}_{chosenComp_12} has TC
42      Add (C^{P_c}_{choseniConf_12}, Conf_R)
43        If true = DelayProperty(C^{P_a}_{xiConf_1}, C^{P_b}_{yiConf_2})
44          TC^{P_ab}_z = GetTC(C^{P_c}_{chosenComp_12})
45          Add TC^{P_ab}_z in TC_LIST
46          UpdateLabelsAndConnect())
47        Else
48          Add TC^{P_ab}_z in TC_LIST
49          for all TC in TC_LIST
50            Add (TC, Conf_R)
51            UpdateLabelsAndConnect()
52      Else goto Stop
53    If case = Stop
54      Add (C^{P_a}_{xiConf_1}, Conf_R)
55      Add (C^{P_b}_{yiConf_1}, Conf_R)
56      UpdateLabelsAndConnect()
```

Listing 8.2: Extended Configuration Folding Algorithm

As described earlier, the configuration folding algorithm compares components in the two configurations in topological ordering. If the components at a particular level fulfil the redundancy conditions, they are folded and added in the folded configuration. In order to incorporate transformation components and delaying of transformation components,

the configuration folding algorithm needs to cater for four cases. Case 1 indicates that the components at a particular level in the two configurations can be folded such that their parametrization is same. Case 2 indicates that the components could be folded but their parametrization is not same. This case also checks if the components at the next higher level have delaying properties or are marked by the configuration developer to allow the delay. If this is the case, the use of transformation component is delayed otherwise the transformation component is introduced and further folding is not possible. Case 3 indicates that the components can be folded and their parametrization is same. This case is evaluated when the components at this level have the delaying properties. This case also checks if the components at the next level also have the delaying properties or are marked to allow the delay. Case 4 indicates that the components can be folded but their parametrization is not same. This case is evaluated when the components at this level have the delaying properties. This case also checks if the components at the next level also have delaying properties or are marked to allow the delay. If this is the case, then folding continues otherwise folding stops as per Stop case shown in the algorithm.

The algorithm makes use of certain functions. The checkFoldingCase(Component comp1, Component comp2) determines which of the above mentioned four cases applies to the two components in the configurations and then process them accordingly. The Add(Component chosen, Configuration folded) adds the chosen component as a result of folding in the folded configuration. The chosen component is determined by evaluating the parameters and relations between their values as discussed previously. UpdateLabelsAndConnect() adjusts the incoming labels of components at the next higher level. These labels are used for making connections between the components in the folded configuration. GetTC(Component chosen) returns the transformation component for the chosen component. The DelayProperty(Component comp1, Component comp2) checks if the components linked to comp1 and comp2 at the next higher level have delaying properties.

8.3 Evaluation

As configuration folding allows energy efficient execution of simultaneously executing context recognition applications, therefore, in order to demonstrate its effectiveness we developed two context recognition applications using our component system. As configuration folding requires runtime analysis of application structures, the component abstraction used by our component system provides a suitable way of performing the analysis. To mention, it is not mandatory for the configuration folding to be applied using the component abstraction only. The concept is general and can be applied using other abstractions as long as those abstractions permit analysis and reconfiguration of application structures in a computationally inexpensive manner.

The two test applications namely the speech recognition and music recognition applications have been implemented using the set of audio features described in [LPL+09]. We used RapidMiner [rap] to train the classifiers for speech and music recognition using several hours of recorded audio data. We used frame size of two second long periods which enabled us to achieve a 90 percent precision using a decision tree classifier. The resulting set of used features are shown in Figure 8.5 for speech and in Figure 8.6 for music.

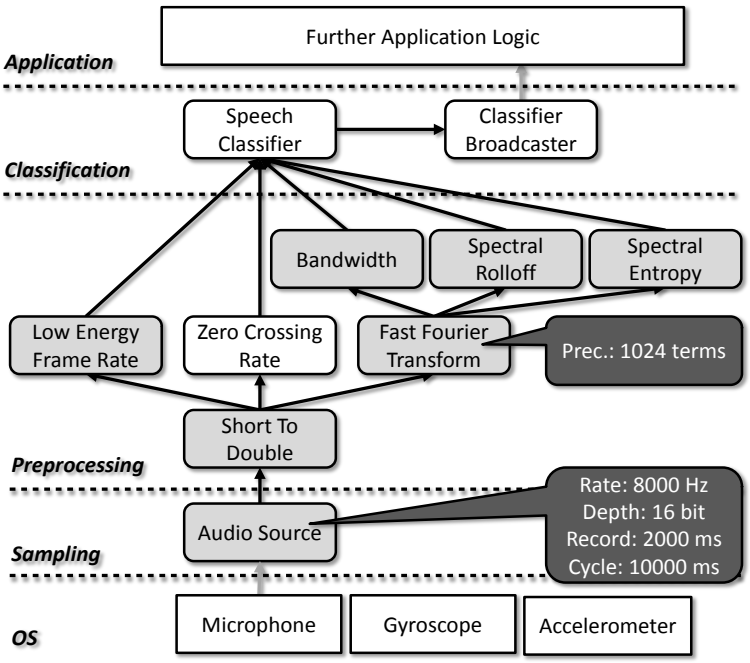

Figure 8.5: Configuration for speech recognition

As depicted in these figures, both classifiers use a varying set of frequency as well as time domain features. As a result, the componentized recognition logic contains a number of identically configured components (marked in grey). The identical components include the actual sampling component (audio source), a conversion component (short to double), the frequency domain transformation (FFT) as well as features from the time domain

(i.e. the low energy frame rate) and the frequency domain (i.e. bandwidth, spectral rolloff, spectral entropy). Besides that, each logic also contains components to compute unique features that are not relevant to the other application (i.e. zero crossing rate, spectral flux). Together, this provides an indication for the optimization potential of configuration folding. The folded configuration for these two configurations is shown in Figure 8.7 and as illustrated, the folded configuration contains only single instance of common components between the two applications, hence enable energy saving by avoiding the redundant computations.

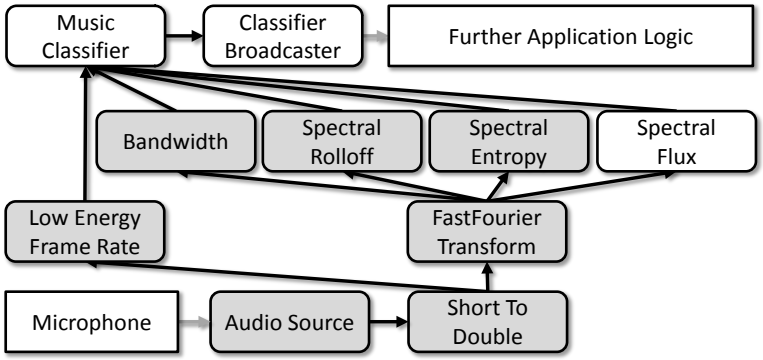

Figure 8.6: Configuration for music recognition

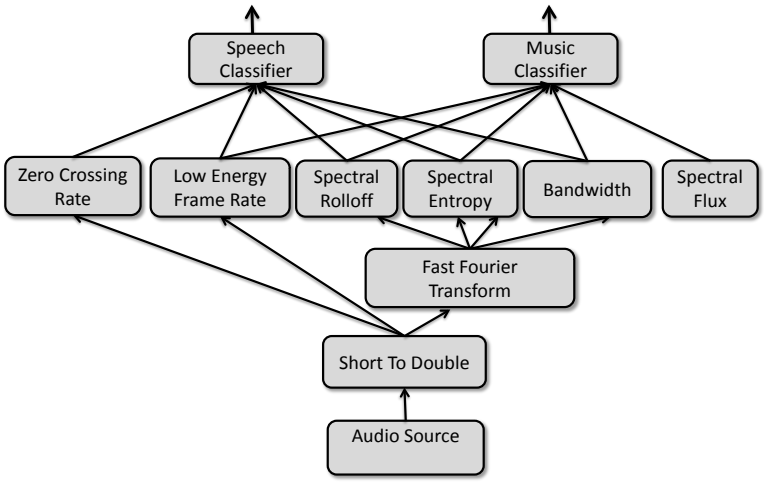

Figure 8.7: Folded configuration for speech and music recognition applications

To evaluate the effectiveness of configuration folding, we performed a detailed experimental analysis of the speech and music recognition applications. As target platform for our experiments, we used a HTC Tattoo which is a low-end smart phone (Qualcomm MSM7225 processor running at 528MHz, 512MB ROM, 256MB RAM) running an Android operating systems in version 1.6. In order to perform precise power measurements,

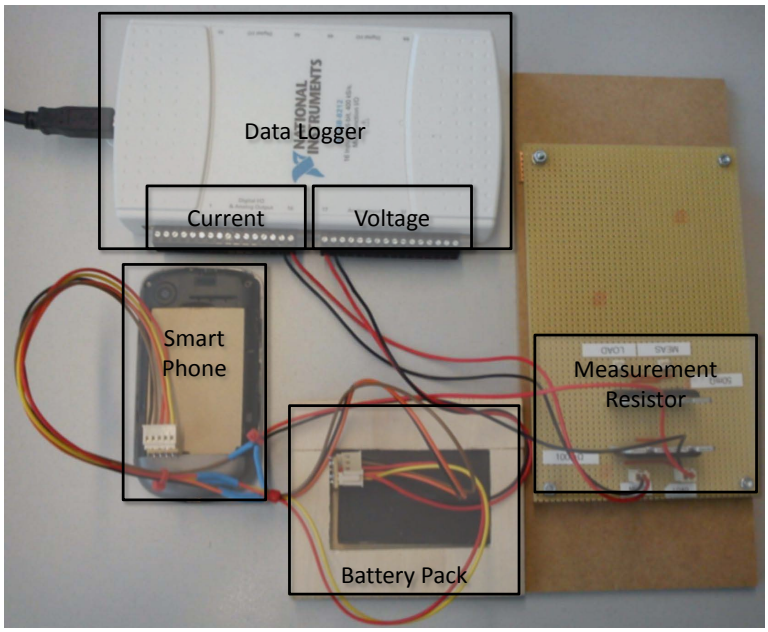

Figure 8.8: Power measurement setup

we rely on a setup that closely follows the one described in [RH10]. As shown in Figure 8.8, this means that we are connecting a high precision measurement resistor of 100 mΩ in series between the battery pack and the HTC Tattoo and we connect both, the battery and the resistor to a high speed data logger (NI USB-6212) to measure the voltage of the battery as well as the power drain of the device. Since the sampling rate of the data logger is limited to 400kHz, we sample both inputs at 200kHz which is slightly lower than the 250kHz measurements performed in [RH10].

Using this setup, we profiled four different software configurations that are depicted in Figure 8.9. In order to improve the readability, the figure shows the average power in Watts (W) aggregated over periods of 100 ms. As a baseline for the power requirements, we measured the power required by the device when the display is running and the screen brightness is on its lowest setting (*idle*) which resulted in an average power drain of 103 mW. In addition, we individually profiled the music (*music*) and the speech (*speech*) recognition application when running on the device in isolation and we profiled the folded configuration (*folded*) that combined both, the music and the speech recognition application. Thereby, we use identical parametrizations for all components to allow the most effective configuration folding that is possible.

As an example, Figure 8.9 shows one processing cycle for each of the tested software configurations over a period of 6 seconds. Using further measurements, we decomposed the power profile of the recognition applications by averaging over the relevant periods as follows. Since the device's display is continuously configured to the lowest brightness setting during all measurements, there is a constant power drain of 103 mW. When the recognition applications begin with their task, this power drain increases by 285 mW for

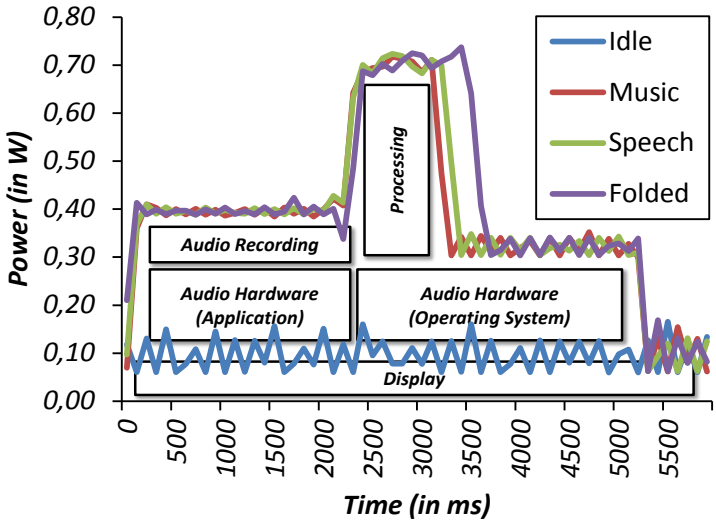

Figure 8.9: Decomposed power usage for speech, music and folded configuration

a period of 2 seconds. This can be attributed to the fact that all recognition applications initially perform audio recording. The 285 mW can be further divided into 219 mW which are required to power the audio hardware and 66 mW which are required for the analogue to digital conversion and the buffering of samples. After the recording is done, all applications begin with the processing which results in an average power drain of 527 mW for the time the computations are performed. However, while decomposing the profiles we found that only 308 mW are actually due to the processor running at full speed. The remaining 219 mW can be attributed to the fact that the HTC Tattoo does not deactivate the audio hardware immediately after recording. Instead, once it is no longer required, it will remain powered on for an additional 3 seconds. Using additional experiments, we found that these 3 seconds are not dependent on the recording time but they appear to be a constant coded into the resource management code of the operating system.

Given these power measurements, we can easily approximate the overall energy requirements for each recognition cycle by determining the computation times required for the individual applications. To do this, we measured the processing times of 100 cycles each which resulted in an average processing time of 1002 ms for music recognition, 1023 ms for speech recognition and 1220 ms for the folded configuration. For all measurements, the standard deviation was well below 10 percent but there were systematic deviations which we attribute primarily to garbage collection that introduces around 100 ms of delay per collection. Note that these timings already show that for an identically parametrized music and speech recognition application, configuration folding saves 40 percent of processing.

To compute the actual energy savings between running the unfolded configurations simultaneously as well as running the folded configuration, however, it is important to consider that the power profile also contains components that cannot be added directly.

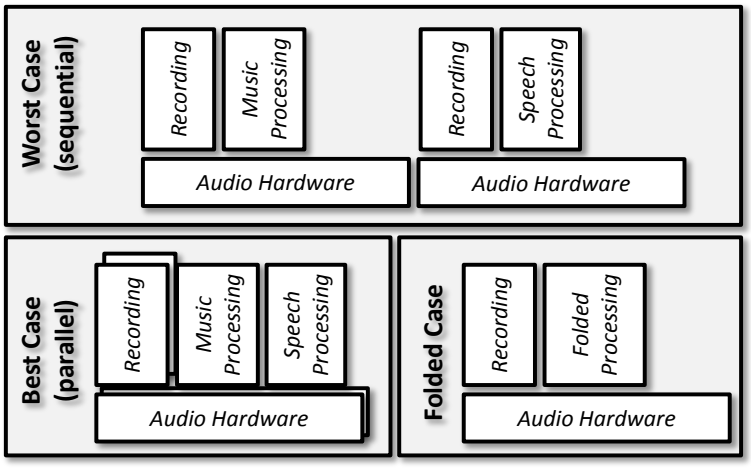

Figure 8.10: Analytical energy model

Thus, it is necessary to consider the interleaving of the music and the speech recognition applications at runtime. The resulting extremes are depicted in Figure 8.10. In the best – that is in the most energy-efficient – case, both applications are starting their cycles at the exact same instance of time. This means that they share the cost for enabling the audio hardware as well as the analogue digital conversion during recording. Consequently, we can compute the energy requirements in this case as $5s \cdot 0.219W + 2s \cdot 0.066W + 2.025s \cdot 0.308W = 1.851J$. In the worst case, both applications are running sequentially which results in an energy usage of $10s \cdot 0.219W + 4s \cdot 0.066W + 2.025s \cdot 0.308W = 3.078J$. In contrast to this, a single cycle of running the folded configuration results requires on average $5s \cdot 0.219W + 2s \cdot 0.066W + 1.220s * 0.308W = 1.603J$ of energy. Thus, the actual energy savings could be as low as 13 and as high as 48 percent. Yet, if we assume that many context recognition applications rather have a low duty cycle in order to conserve resources, however, we can expect the savings to be closer to 48 than 13 percent, in practice.

To determine the effect of the configuration folding algorithm, we performed a similar set of measurements on the algorithm itself. This resulted in an average power usage of 448 mW for 1739 ms which results in 0.78 J for one run. From this effort, 865 ms and 724 ms can be attributed to configuration loading and graph transformation, respectively. The remaining 150 ms are required to execute the configuration folding algorithm. Thus, the major overheads will only be experienced once – on application startup. However, even when including these overheads, given the potential energy savings of 0.248 J or 1.475 J per cycle, the 0.78 J required by the algorithm are amortized after 1 to 3 cycles. Note that if the applications operate on a low duty cycle, the lower number of cycles would be more likely whereas a high duty cycle would reduce the amount of time required to break even.

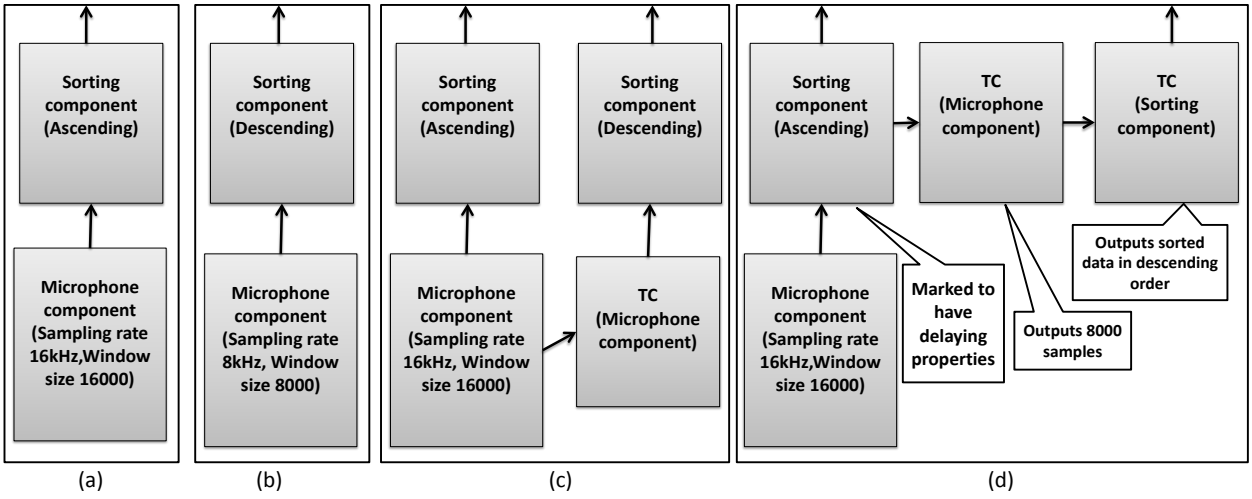

Figure 8.11: Configurations used in the evaluation of extended configuration folding

8.3.1 Parameter Handling

The evaluation of speech and music recognition applications in the previous section does not involve handling of parametrization. In this section we evaluate how much energy savings are possible if we follow the the handling of parametrization in configuration folding as described earlier. As the proposed solution for handling parameters involves use of transformation component and delay of use of transformation component in a folded configuration, we consider two example configurations consisting of two components each with different parametrization. The first configuration shown in Figure 8.11(a) consists of a microphone sensing component with sampling rate of 16kHz and output window size of 16000 samples. The second component in this configuration is a sorting component which sorts the incoming audio data samples in ascending order. The second configuration Figure 8.11(b) consists of a microphone sampling component with sampling rate of 8kHz and output window size of 8000 samples. The sorting component in this configuration is parametrized to perform sorting of incoming audio data in descending order. The sorting component is (optionally) marked by the configuration developer (see Section 8.2.3) to have delaying properties in the configurations though it does not have Property 2 as it operates on multiple input values.

To determine the possible gains of different scenarios, we used a SAMSUNG Galaxy Nexus which is a high-end smart phone (TI OMAP 4460 chipset, Dual-core 1.2 GHz Cortex-A9 cpu and 1 GB RAM) running Android 4.2 as our target platform. Using the precise energy measurement set shown in Figure 8.8, we are able to capture both the voltage and current at 200KHz. All measurements were made when the phone was running on minimum brightness and radio transceivers off. The average power consumption for this base condition is 604mW. In addition, we add low-level instrumentation to our component system to measure the total sampling and processing time for an individual window and we capture the average time per window over 20 consecutive windows.

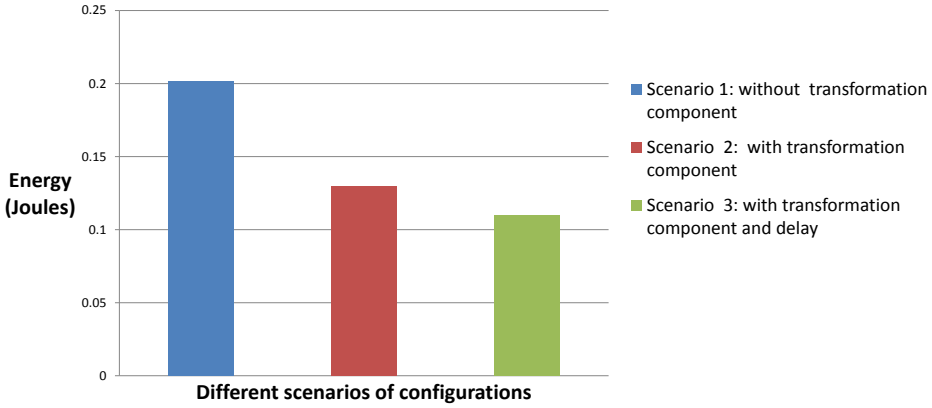

Figure 8.12: Energy consumption of scenarios for extended configuration folding

In scenario 1, we run the two configurations individually, i.e without applying configuration folding. This scenario is depicted in Figure 8.11(a) and (b). In this scenario the average power drain for the first configuration was 690 mW and the average time for sampling and processing one window was 1046 msec (i.e, 1 sec sampling + 46 msec processing). The average power drain for the second configuration was 711 mW and the average time was 1045 msec/window.

In scenario 2, we apply configuration folding to these two configurations and due to the difference in parametrization, a transformation component is also used in the folded configuration. This transformation component forwards every second value to the sorting component. Thereby, we do not mark the sorting component to exhibit the delaying property so that the transformation component in the folded configuration is inserted directly after the sampling component as depicted in Figure 8.11(c). In this scenario, the average power drain was 728mW with an average time of 1048 msec/window.

In scenario 3, we apply configuration folding to the two configurations. The resulting configuration is depicted in Figure 8.11(d). For this scenario, we marked the sorting component to exhibit the delaying property. As a result the folding algorithm introduces the transformation component for the sampling component after the sorting component. Since the sorting components in the two configurations also exhibit a different parametrization, a second transformation component is added for them. This transformation component simply inverts the order of the values in the window in order to account for the differences in sorting direction. In this scenario, the average power consumption for the folded configuration was 709mW with an average time of 1046 msec/window.

In order to measure the actual energy gains depicted in Figure 8.12, we subtract the base power drain from the power drain of above mentioned cases. This leads to energy usage for the scenario 1 to be (690-604)mW *1046 msec = 0.08996 Joules for the first configuration and (711-604)mW *1045 msec = 0.1118 Joules for the second configuration. Similarly the energy usage for the scenario 2 was computed to be (728-604)mW *1048 msec = 0.1299 Joules and (709-604)mW *1046 msec = 0.1098 Joules for scenario 3.

In order to compute the energy savings, we add the energy usage of the two configurations in scenario 1. This is done because in this scenario the configuration folding is not applied and therefore simultaneously executing them could result in the addition of their respective energy consumptions. It is noteworthy that this reflects the worst case, i.e., where both applications are running independently. Depending on the interleaving of the applications, the actual power consumption might be less. For a more detailed discussion and analysis, we refer to our analysis of basic folding algorithm earlier in the chapter which explicitly evaluates the effect of interleaving. As a result, the total energy consumption for scenario 1 is 0.08996+0.1118= 0.20176 Joules. Therefore, the total savings in the energy consumption between scenario 1 and scenario 2 is computed to be 35.5% and from scenario 1 and scenario 3 is 45.5%. The savings between scenario 2 and scenario 3 are computed to be 15.4%. The energy consumptions for the algorithm for scenario 2 and scenario 3 were computed to be (1777.98mW * 5msec) = 0.005 Joules and (1245.7mW * 6msec) = 0.0074 Joules respectively.

The energy measurements of configuration folding with and without parametrization show considerable energy savings when applied on the test applications. To mention, the energy savings depend upon the level of redundancy, complexity of components, type of parameters used in the configurations. For applications which have a large number of redundant components as per the definition of redundancy described for configuration folding, the energy savings will be large. Similarly if complexity of redundant components is high, the savings are more and in the same way if the redundant components are parametrized with the commonly used parameters, there will be a potential for energy savings.

8.3.2 Requirements Coverage

One of the four design requirements for our framework is efficiency. The requirement on efficiency is described at two levels namely the efficiency for creating the context recognition applications and efficiency for executing these applications in an energy efficiency manner. Configuration folding provides the latter one. It allows energy efficient execution of context recognition applications executing simultaneously by finding and removing redundancies between them. This is clearly shown in the examples of speech and sound recognition applications discussed above. The associated cost of running configuration folding and the potential savings shows that configuration folding is a viable approach for executing multiple applications simultaneously. The handling of parametrization using transformation components also extends the applicability of folding to scenarios where the applications differ in parametrization. The identification of commonly used parameters and relations for their values to ensure the correctness of the output of the folded configuration provides developers and users guarantee of receiving the desired context recognition as they would have expected from running the applications standalone. Looking at the results of our experiments above, we can say that configuration folding adds support for energy efficiency to our framework and therefore aids in fulfilling the design requirement on efficiency.

8.4 Summary

In this chapter we have described configuration folding in detail. We first formalized the problem that configuration folding solves and then presented the basic folding algorithm. The chapter also discussed the effects of differences in parametrization on configuration folding and discussed the use of transformation components for handling it. Towards this end, the chapter discussed commonly used parameters in context recognition applications and discussed relations between their possible values. In order to support the delaying of transformation components in the folded configuration, the chapter described two approaches at the component level and at the configuration level. After that, the chapter provided the listing of extended version of the configuration folding algorithm. Finally, the chapter concluded by evaluating configuration folding and showing the potential energy savings it can provide to fulfil the framework's requirement on efficiency.

9 CONCLUSIONS AND OUTLOOK

In this thesis we have presented our framework for generic and energy efficient context recognition for smart phones. The two features i.e. generic and energy efficiency are essential based on the challenges posed by the growing number of context recognition applications and the increasing computational requirements of these applications on smart phones. By generic context recognition we mean that using our framework, developers are able to develop applications for any context. Consequently, it provides a single platform for their execution which also opens possibilities to incorporate energy efficiency solutions at the system level so that different applications can benefit from them. By energy efficient context recognition we mean that when multiple context recognition applications are executed using a same platform, the system can identify and remove functional redundancies between them. It also means that developers can model their applications in such a way that the applications can take advantage of different existing energy efficiency techniques through their generic applicability supported by the framework.

9.1 Conclusions

We have based the design of our framework on the existing state of the art on context recognition applications, frameworks and energy efficiency techniques. The analysis of the existing work has revealed that in order to create a generic and energy efficient solution, our framework must fulfil four design requirements. These requirements are uniformity, extensibility, configurability and efficiency. We have defined uniformity as the ability to provide access to context information required by different applications in a consistent way which in turn requires applications to be developed using a same design approach. We have defined extensibility as the ability to extend the existing set of context recognition methods to ensure that the framework is not closed in terms of support for new applications. We have defined configurability as the ability to change the settings of applications to enable different behaviour or different performance. We have defined efficiency at two levels namely the efficiency for creating the applications and efficiency for achieving energy efficient execution of applications. In order to fulfil these requirements, we developed our framework consisting of two sub-systems namely the component system and the activation system supported with off line development tools and energy efficiency techniques. The detailed description for these systems and techniques has been given in their respective chapters. Next, we summarize them briefly to show how they fulfill the four design requirements collectively.

The component system uses a component abstraction to model context recognition applications. The component abstraction used by the component system consists of components, connectors and configurations. The applications acquire context information using set of components and connectors called configuration. Hence, a configuration represents the context recognition logic for the applications. The runtime system of the component system loads configurations from different applications and instantiate the required components and connections between the components. The component system provides a single platform for execution of applications targeting different contexts developed using a same design approach. Consequently, it enables applications to access the context information in a uniform manner. This fulfils design requirement on uniformity. The component system provides a component toolkit which consists of a number of components which can be reused to create new applications. Also, new components can be added to the component toolkit. Moreover, the component model enables developers to make changes in the configurations by adding or removing components and adjusting the connections between them accordingly. Thus, the provisioning of the component toolkit fulfils design requirements on extensibility. The components in the component system are parametrizable meaning that developers can use different values for the parameters to enable different functionality or different performance from same components. Thus, the provisioning of parameters supports configurability. Lastly, the component system is equipped with a graphical editor for creating configurations to be used by the applications. In addition, the graphical editor is equipped with code generation and validation utilities so that the generated code for the configurations can be directly use by the applications enabling rapid prototyping. Thus, provisioning of graphical editor, its associated tools and the component toolkit fulfils the design requirement on efficiency for creating the applications.

The activation system sits on top of the activation system and uses a state machine abstraction for enabling context dependent activation of configurations associated with an application. The state machine model used by the activation system consists of states and transitions. A state represents a step in the overall recognition logic used by the application. A state is associated with configurations. The configurations are created and executed using the component system. Hence, there is a strong dependency of activation system on the component system. A transition represents a change of state based on certain conditions. A condition is represented by rule and evaluated using the rule engine of the activation system. Therefore, at any time instance only the configurations attached to one state are executed thereby providing energy efficient execution compared to the case when all the configurations are executed all the time whether they are required or not. The activation system not only provides context dependent execution of required configurations but it also enables generic applicability of other energy efficiency techniques namely Suppression, Substitution, Adaptation and Piggybacking. Similar to the component system, the off line development tools of activation system aid rapid prototyping of state machines by using graphical editor and code generation utilities. Hence, we can say that the functionalities provided by the activation system fulfil the design requirement on efficiency for creating the applications and also enable their energy efficient execution.

The framework is equipped with a novel energy efficiency technique called configuration folding. Using configuration folding the framework identifies redundant components between simultaneously executing configurations of different applications. The outcome is a single folded configuration which is suitable for all the applications. We have also investigated means to improve the impact of configuration folding when the redundant components between the applications differ in parametrizations. In order to achieve this, we analysed different applications that we have created during different projects. Our analysis showed that there are certain parameters which are commonly used in most of the applications. We identified those parameters and described their semantics and identified possible relations between their values so that configuration folding can be applied. Our solution to parametrization problem was the use of transformation components. We also discussed two properties required by components in order to delay the use of transformation components in a folded configuration to achieve more savings. The evaluations of configuration folding on the test applications showed significant energy savings. Depending upon the degree of redundancy, the actual savings between other applications can be different. Based on the experiments and the obtained results we can say that configuration folding is a suitable energy efficiency approach when multiple applications are executed simultaneously, hence it fulfils the design requirement on energy efficiency.

We have also developed a number of applications using our framework. These applications have been developed as part of different European projects in health and transportation domains. Consequently, the created applications covered a broad range of contexts including speech recognition, location recognition, health monitoring, activity recognition and transportation. Moreover, our framework is currently being used in other European projects in which it is used for creating components and configurations for energy and smart meter domains.

9.2 Outlook

By using component abstraction, state machine abstraction and configuration folding, this thesis has contributed a generic and energy efficient context recognition framework for smart phones. The work presented in thesis has moved forward the work towards fulfilling the vision of ubiquitous computing presented by Mark Weiser. Though our solution fulfils its objectives and design requirements, nevertheless it also highlights directions for further investigations. In the following we identify possible opportunities for future work.

Firstly, the energy efficiency solutions presented in this thesis are implemented and evaluated for smart phones whereas solutions for ubiquitous computing environments should also provide means for energy efficient execution of services related to context recognition in distributed environments. Therefore, applicability of configuration folding with and without parametrization in distributed settings should be investigated. Possible use case for this could be environments where users share same context. Therefore, it might be possible that instead of doing same computations on every user device, the computations are done once and results are shared between different users. However, doing so would

also involve privacy and trust concerns between users i.e. why users should trust each other and rely on the context information gathered from a different device and if doing so affects their privacy. In addition, energy cost of communicating the context between users or between users and other entities will have to be taken in to account. The affect of these costs can be significantly high if the environment is highly dynamic.

Secondly, in the current implementation of the framework, the development of configurations and state machines is static i.e. once created they can be changed again only by the developers themselves. It would be interesting to introduce self optimization functionality in the framework such that the configurations and state machines are automatically optimized based on user′s changing routines or other internal and external factors such as availability of resources. The self optimization of configurations and state machines would require introduction of semantics about different contexts and defining of relations between the semantics such that self optimization does not break the original objectives of configurations and state machines.

Thirdly, the current implementation of the framework is targeted at Android based devices. Also, the graphical editors are implemented as plug-ins for Eclipse IDE. Since Android and iOS based hand-held devices are the fastest growing in numbers, the framework should also be implemented for iOS based devices for wider coverage of users. This would not only require implementation of runtime systems but would also require porting of our existing component toolkit. Moreover, to further support rapid prototyping, the graphical editors will also have to be implemented for other IDEs such as Xcode and AndroidStudio.

Finally, in the extended configuration folding algorithm there is a possibility for further optimization. The optimization possibility deals with the case when a folded configuration has more than one transformation components. The question here is how to arrange the transformation components in the most energy efficient sequence. As an example consider a folded configuration which has a transformation component for audio sensor and transformation component for arithmetic component. The transformation component for audio sensor forwards every second value from its input and transformation component for the arithmetic component first remove a certain value from every input and then adds a certain value to the input. Clearly, if the transformation component for the audio component is used first then the transformation component for the arithmetic component would have to process half of the values compared to the case when arithmetic transformation component is applied before the audio transformation component. To perform such optimization would require automated analysis of computational complexities of transformation components and identifying their most energy efficient sequence.

Bibliography

[AHIM13] W. Apolinarski, M. Handte, M.U. Iqbal, and P.J. Marron. Pike: Enabling secure interaction with piggybacked key-exchange. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, pages 94–102, March 2013.

[AHIM14] Wolfgang Apolinarski, Marcus Handte, Muhammad Umer Iqbal, and Pedro José Marrón. Secure interaction with piggybacked key-exchange. *Pervasive Mob. Comput.*, 10:22–33, February 2014.

[AIP14] W. Apolinarski, U. Iqbal, and J.X. Parreira. The gambas middleware and sdk for smart city applications. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*, pages 117–122, March 2014.

[anda] Android. `http://developer.android.com/`.

[Andb] Android Activity. `http://developer.android.com/training/basics/activity-lifecycle/starting.html`.

[Andc] Android Intent Filters. `http://developer.android.com/guide/components/intents-filters.html`.

[Andd] Android Manifest. `http://developer.android.com/guide/topics/manifest/manifest-intro.html`.

[Ande] Android Service. `http://developer.android.com/guide/components/services.html`.

[App] Apple iOS. `https://www.apple.com/ios/`.

[ARK+06] M. Alwan, P.J. Rajendran, S. Kell, D. Mack, S. Dalal, M. Wolfe, and R. Felder. A smart and passive floor-vibration based fall detector for elderly. In *Information and Communication Technologies, 2006. ICTTA '06. 2nd*, volume 1, pages 1003–1007, 2006.

[BAPH09] Fehmi Ben Abdesslem, Andrew Phillips, and Tristan Henderson. Less is more: Energy-efficient mobile sensing with senseless. In *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*, MobiHeld '09, pages 61–62, New York, NY, USA, 2009. ACM.

[Bar04a] Jakob E. Bardram. Applications of context-aware computing in hospital work: Examples and design principles. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1574–1579, New York, NY, USA, 2004. ACM.

[Bar04b] Jakob E. Bardram. Applications of context-aware computing in hospital work: Examples and design principles. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1574–1579, New York, NY, USA, 2004. ACM.

[bes] Besos: Building Energy Decision Support Systems for Smart Cities. `http://besos-project.eu`.

[BHSR04] C. Becker, M. Handte, G. Schiele, and K. Rothermel. Pcom - a component system for pervasive computing. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pages 67–76, March 2004.

[BI04] Ling Bao and StephenS. Intille. Activity recognition from user-annotated acceleration data. In Alois Ferscha and Friedemann Mattern, editors, *Pervasive Computing*, volume 3001 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2004.

[BKLA06] David Bannach, Kai Kunze, Paul Lukowicz, and Oliver Amft. Distributed modular toolbox for multi-modal context recognition. In Werner Grass, Bernhard Sick, and Klaus Waldschmidt, editors, *Architecture of Computing Systems - ARCS 2006*, volume 3894 of *Lecture Notes in Computer Science*, pages 99–113. Springer Berlin Heidelberg, 2006.

[BLA08] D. Bannach, P. Lukowicz, and O. Amft. Rapid prototyping of activity recognition applications. *Pervasive Computing, IEEE*, 7(2):22–31, April 2008.

[BP00] P. Bahl and V.N. Padmanabhan. Radar: an in-building rf-based user location and tracking system. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 775–784 vol.2, 2000.

[BP07] Ari Y. Benbasat and Joseph A. Paradiso. A framework for the automated generation of power-efficient classifiers for embedded sensor nodes. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 219–232, New York, NY, USA, 2007. ACM.

[BvdVG+10] A.K. Bourke, P. van de Ven, M. Gamble, R. O'Connor, K. Murphy, E. Bogan, E. McQuade, P. Finucane, G. OLaighin, and J. Nelson. Assessment of waist-worn tri-axial accelerometer based fall-detection algorithms using continuous unsupervised activities. In *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pages 2782–2785, Aug 2010.

[CH10] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIX annual technical conference*, pages 1–14, 2010.

[Com] Common sense toolkit (cstk). `http://cstk.sourceforge.net//`.

[Dra] Draw2d. `http://www.eclipse.org/gef/draw2d/`.

[Ecla] Eclipse. `http://www.eclipse.org/`.

[Eclb] Eclipse Modelling Framework. `http://www.eclipse.org/modeling`.

[Eclc] Graphical Editing Framework. `http://eclipse.org/gef`.

[EML+07] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G-S. Ahn, and A. T. Campbell. The bikenet mobile sensing system for cyclist experience mapping. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 87–101, New York, NY, USA, 2007. ACM.

[emt] Emt: Empresa municipal de transportes de madrid. `http://www.emtmadrid.es/`.

[EPMK08] Miikka Ermes, Juha Parkka, Jani Mantyjarvi, and Ilkka Korhonen. Detection of daily activities and sports with wearable sensors in controlled and uncontrolled conditions. *Information Technology in Biomedicine, IEEE Transactions on*, 12(1):20–26, 2008.

[Fac] Facebook. `http://www.facebook.com/`.

[FCC+07] Jon Froehlich, Mike Y. Chen, Sunny Consolvo, Beverly Harrison, and James A. Landay. Myexperience: A system for in situ tracing and capturing of user feedback on mobile phones. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 57–70, New York, NY, USA, 2007. ACM.

[FCCRS12] Antonio Fernndez-Caballero, Jos Carlos Castillo, and Jos Mara Rodrguez-Snchez. Human activity monitoring by local and global finite state machines. *Expert Systems with Applications*, 39(8):6982 – 6993, 2012.

[FGR+12] B. Flipsen, J. Geraedts, A. Reinders, C. Bakker, I. Dafnomilis, and A. Gudadhe. Environmental sizing of smartphone batteries. In *Electronics Goes Green 2012+ (EGG), 2012*, pages 1–9, Sept 2012.

[gam] GAMBAS (Generic Adaptive Middleware for Behavior-driven Autonomous Services). `http://www.gambas-ict.eu/`.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[GJS08] M. Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 321–330, May 2008.

[goo] Google. `http://www.google.com/`.

[GSBB10] Henning Guenther, Firas El Simrany, Martin Berchtold, and Michael Beigl. A tool chain for a lightweight, robust and uncertainty-based context classification system (ccs). In *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*, pages 1–9, Feb 2010.

[HBS03] Marcus Handte, Christian Becker, and Gregor Schiele. Experiences: Minimalism and Extensibility in BASE. In *Proceedings of the System Support for Ubiquitous Computing Workshop (Ubisys03) at the Fifth Annual Conference on Ubiquitous Computing (Ubicomp03); Seattle, Washington, October 12, 2003*, pages 1–8. online proceedings, October 2003.

[HIA+10] M. Handte, U. Iqbal, Wolfgang Apolinarski, S. Wagner, and P.J. Marron. The narf architecture for generic personal context recognition. In *Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC), 2010 IEEE International Conference on*, pages 123–130, June 2010.

[HIAM10] Marcus Handte, Umer Iqbal, Wolfgang Apolinarski, and Pedro José Marrón. Challenges in ubiquitous context recognition with personal mobile devices. In *Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems*, CASEMANS '10, pages 6:40–6:45, New York, NY, USA, 2010. ACM.

[HIW+14] Marcus Handte, Muhammad Umer Iqbal, Stephan Wagner, Wolfgang Apolinarski, Pedro José Marrón, Eva Maria Muñoz Navarro, Santiago Martinez, Sara Izquierdo Barthelemy, and Mario González Fernández. Crowd density estimation for public transport vehicles. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014.*, pages 315–322, 2014.

[HKAK08] Yu-Jin Hong, Ig-Jae Kim, Sang Chul Ahn, and Hyoung-Gon Kim. Activity recognition using wearable sensors for elder care. In *Future Generation Communication and Networking, 2008. FGCN'08. Second International Conference on*, volume 2, pages 302–305. IEEE, 2008.

[HNT13] Samuli Hemminki, Petteri Nurmi, and Sasu Tarkoma. Accelerometer-based transportation mode detection on smartphones. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 13:1–13:14, New York, NY, USA, 2013. ACM.

[HTGT13] Seyed Amir Hoseini-Tabatabaei, Alexander Gluhak, and Rahim Tafazolli. A survey on smartphone-based systems for opportunistic user context recognition. *ACM Comput. Surv.*, 45(3):27:1–27:51, July 2013.

[IFW+13] Muhammad Umer Iqbal, Ngewi Fet, Stephan Wagner, Marcus Handte, and Pedro José Marrón. Living++: A platform for assisted living applications. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, UbiComp '13 Adjunct, pages 853–860, New York, NY, USA, 2013. ACM.

[IHM15] Muhammad Umer Iqbal, Marcus Handte, and Pedro José Marrón. A generic approach for energy efcient context recognition using smart phones. In *Intelligent Environments (IE), 2015 11th International Conference on*. IEEE, 2015.

[IHW+12a] M.U. Iqbal, M. Handte, S. Wagner, Wolfgang Apolinarski, and P.J. Marron. Configuration folding: An energy efficient technique for context recognition. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on*, pages 519–521, March 2012.

[IHW+12b] M.U. Iqbal, M. Handte, S. Wagner, Wolfgang Apolinarski, and P.J. Marron. Enabling energy-efficient context recognition with configuration folding. In *Pervasive Computing and Communications (PerCom), 2012 IEEE International Conference on*, pages 198–205, March 2012.

[Jav] Java. `http://www.java.com/`.

[JLT04] H. Junker, P. Lukowicz, and G. Troster. Sampling frequency, signal resolution and the accuracy of wearable context recognition systems. In *Wearable Computers, 2004. ISWC 2004. Eighth International Symposium on*, volume 1, pages 176–177, Oct 2004.

[JLY+12] Younghyun Ju, Youngki Lee, Jihyun Yu, Chulhong Min, Insik Shin, and Junehwa Song. Symphoney: A coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Proceedings of the 10th ACM*

Conference on Embedded Network Sensor Systems, SenSys '12, pages 211–224, New York, NY, USA, 2012. ACM.

[Kah62] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.

[KKES10] Donnie H. Kim, Younghun Kim, Deborah Estrin, and Mani B. Srivastava. Sensloc: Sensing everyday places and paths using less energy. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 43–56, New York, NY, USA, 2010. ACM.

[KLGT09] Mikkel Baun Kjærgaard, Jakob Langdal, Torben Godsk, and Thomas Toftkjær. Entracked: Energy-efficient robust position tracking for mobile devices. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 221–234, New York, NY, USA, 2009. ACM.

[KLJ+08] Seungwoo Kang, Jinwon Lee, Hyukjae Jang, Hyonik Lee, Youngki Lee, Souneil Park, Taiwoo Park, and Junehwa Song. Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 267–280, New York, NY, USA, 2008. ACM.

[KMD13] Immanuel Koenig, Abdul Qudoos Memon, and Klaus David. Energy consumption of the sensors of smartphones. In *Wireless Communication Systems (ISWCS 2013), Proceedings of the Tenth International Symposium on*, pages 1–5, Aug 2013.

[Lan91] D. Landgrebe. A survey of decision tree classifier methodology. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(3):660–674, May 1991.

[LBBP+11] Hong Lu, A.J. Bernheim Brush, Bodhi Priyantha, AmyK. Karlson, and Jie Liu. Speakersense: Energy efficient unobtrusive speaker identification on mobile phones. In Kent Lyons, Jeffrey Hightower, and ElaineM. Huang, editors, *Pervasive Computing*, volume 6696 of *Lecture Notes in Computer Science*, pages 188–205. Springer Berlin Heidelberg, 2011.

[LCB06] Jonathan Lester, Tanzeem Choudhury, and Gaetano Borriello. A practical approach to recognizing physical activities. In *Proceedings of the 4th International Conference on Pervasive Computing*, PERVASIVE'06, pages 1–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[liv] LIVING++(TV-based Assisted Living System for Elderly to live well and independently). `http://www.livingplusplus.eu/`.

[LL13] Oscar D Lara and Miguel A Labrador. A survey on human activity recognition using wearable sensors. *Communications Surveys & Tutorials, IEEE*, 15(3):1192–1209, 2013.

[LML+10] N.D. Lane, E. Miluzzo, Hong Lu, D. Peebles, T. Choudhury, and A.T. Campbell. A survey of mobile phone sensing. *Communications Magazine, IEEE*, 48(9):140–150, Sept 2010.

[LPL+09] Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. Soundsense: Scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 165–178, New York, NY, USA, 2009. ACM.

[LSDM01] Dongge Li, Ishwar K. Sethi, Nevenka Dimitrova, and Tom McGee. Classification of general audio data for content-based retrieval. *Pattern Recogn. Lett.*, 22(5):533–544, April 2001.

[LY13] Miguel A. Labrador and Oscar D. Lara Yejas. *Human Activity Recognition: Using Wearable Sensors and Smartphones*. Chapman & Hall/CRC, 2013.

[LZY+12] Yunji Liang, Xingshe Zhou, Zhiwen Yu, Bin Guo, and Yue Yang. Energy efficient activity recognition based on low resolution accelerometer in smart phones. In Ruixuan Li, Jiannong Cao, and Julien Bourgeois, editors, *Advances in Grid and Pervasive Computing*, volume 7296 of *Lecture Notes in Computer Science*, pages 122–136. Springer Berlin Heidelberg, 2012.

[MEBH08] M Mun, Deborah Estrin, Jeff Burke, and Mark Hansen. Parsimonious mobility classification using gsm and wifi traces. In *Proceedings of the Fifth Workshop on Embedded Networked Sensors (HotEmNets)*, 2008.

[MGP+12] Yehya Mohamad, Henrike Gappa, Jaroslav Pullmann, Gaby Nordbrock, Carlos Velasco, Marcus Handte, Stephan Wagner, and Marcel Schweda. Context-aware support for people with dementia and their families. In *Deutscher AAL-Kongress*, 2012.

[MLEC07] Emiliano Miluzzo, Nicholas D. Lane, Shane B. Eisenman, and Andrew T. Campbell. Cenceme: Injecting sensing presence into social networking applications. In *Proceedings of the 2Nd European Conference on Smart Sensing and Context*, EuroSSC'07, pages 1–28, Berlin, Heidelberg, 2007. Springer-Verlag.

[MM01] A. Marcus and J.I. Maletic. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pages 107–114, Nov 2001.

[MPR08] Prashanth Mohan, Venkata N. Padmanabhan, and Ramachandran Ramjee. Nericell: Rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 323–336, New York, NY, USA, 2008. ACM.

[MSTC94] Donald Michie, D. J. Spiegelhalter, C. C. Taylor, and John Campbell, editors. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, Upper Saddle River, NJ, USA, 1994.

[MWK+06] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. Yale: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 935–940, New York, NY, USA, 2006. ACM.

[Mys] MySpace. `http://www.myspace.com/`.

[Nat12] Suman Nath. Ace: Exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[NKL+07] Petteri Nurmi, Joonas Kukkonen, Eemil Lagerspetz, Jukka Suomela, and Patrik Floréen. Betelgeuse: A tool for bluetooth data gathering. In *Proceedings of the ICST 2Nd International Conference on Body Area Networks*, BodyNets '07, pages 21:1–21:8, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[PFW11] G.P. Perrucci, F.H.P. Fitzek, and J. Widmer. Survey on energy consumption entities on the smartphone platform. In *Vehicular Technology Conference (VTC Spring), 2011 IEEE 73rd*, pages 1–6, May 2011.

[PHZ12] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[PLL11] B. Priyantha, D. Lymberopoulos, and Jie Liu. Littlerock: Enabling energy-efficient continuous sensing on mobile phones. *Pervasive Computing, IEEE*, 10(2):12–15, April 2011.

[Qui86] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[rap] RapidMiner: Data mining tool. `https://rapidminer.com/`.

[Rey02] D.A. Reynolds. An overview of automatic speaker recognition technology. In *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, volume 4, pages IV–4072–IV–4075, May 2002.

[RH10] A. Rice and S. Hay. Decomposing power measurements for mobile devices. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 70–78, March 2010.

[RJ93] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of Speech Recognition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[RMB+10] Sasank Reddy, Min Mun, Jeff Burke, Deborah Estrin, Mark Hansen, and Mani Srivastava. Using mobile phones to determine transportation modes. *ACM Trans. Sen. Netw.*, 6(2):13:1–13:27, March 2010.

[ROPT05] M. Raento, A. Oulasvirta, R. Petit, and H. Toivonen. Contextphone: a prototyping platform for context-aware mobile applications. *Pervasive Computing, IEEE*, 4(2):51–59, Jan 2005.

[RPKL12] Moo-Ryong Ra, Bodhi Priyantha, Aman Kansal, and Jie Liu. Improving energy efficiency of personal sensing applications with heterogeneous multiprocessors. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 1–10, New York, NY, USA, 2012. ACM.

[SAW94] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90, Dec 1994.

[SBCR05] Jacob Sorber, Nilanjan Banerjee, Mark D. Corner, and Sami Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 261–274, New York, NY, USA, 2005. ACM.

[SBG99] Albrecht Schmidt, Michael Beigl, and Hans-W Gellersen. There is more to context than location. *Computers & Graphics*, 23(6):893 – 901, 1999.

[SBS02] Eugene Shih, Paramvir Bahl, and Michael J. Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, pages 160–171, New York, NY, USA, 2002. ACM.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 434–441, New York, NY, USA, 1999. ACM.

[sma] Smart grid key neighborhood indicator cockpit (smartkye). `http://smartkye.eu/`.

[SS97] E. Scheirer and M. Slaney. Construction and evaluation of a robust multifeature speech/music discriminator. In *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, volume 2, pages 1331–1334 vol.2, Apr 1997.

[ST09] F. Sposaro and G. Tyson. ifall: An android application for fall monitoring and response. In *Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE*, pages 6119–6122, Sept 2009.

[SVL+06] Timothy Sohn, Alex Varshavsky, Anthony LaMarca, Mike Y. Chen, Tanzeem Choudhury, Ian Smith, Sunny Consolvo, Jeffrey Hightower, William G. Griswold, and Eyal de Lara. Mobility detection using everyday gsm traces. In *Proceedings of the 8th International Conference on Ubiquitous Computing*, UbiComp'06, pages 212–224, Berlin, Heidelberg, 2006. Springer-Verlag.

[SWP+09] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[SZG+14] Kartik Sankaran, Minhui Zhu, Xiang Fa Guo, Akkihebbal L Ananda, Mun Choon Chan, and Li-Shiuan Peh. Using mobile phone barometer for low-power transportation context detection. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, pages 191–205. ACM, 2014.

[TBGE10] Arvind Thiagarajan, James Biagioni, Tomas Gerlich, and Jakob Eriksson. Cooperative transit tracking using smart-phones. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 85–98, New York, NY, USA, 2010. ACM.

[TC02] G. Tzanetakis and P. Cook. Musical genre classification of audio signals. *Speech and Audio Processing, IEEE Transactions on*, 10(5):293–302, Jul 2002.

[TJDS09] T. Teixeira, Deokwoo Jung, G. Dublon, and A. Savvides. Recognizing activities from context and arm pose using finite state machines. In *Distributed*

Smart Cameras, 2009. ICDSC 2009. Third ACM/IEEE International Conference on, pages 1–8, Aug 2009.

[TRL+09] Arvind Thiagarajan, Lenin Ravindranath, Katrina LaCurts, Samuel Madden, Hari Balakrishnan, Sivan Toledo, and Jakob Eriksson. Vtrack: Accurate, energy-aware road traffic delay estimation using mobile phones. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 85–98, New York, NY, USA, 2009. ACM.

[VBH03] Marc A Viredaz, Lawrence S Brakmo, and William R Hamburgen. Energy management on handheld devices. *Queue*, 1(7):44–52, October 2003.

[Wal96] James S Walker. *Fast fourier transforms*, volume 24. CRC press, 1996.

[Wei99] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999.

[wet] Wetter.com. `http://wetter.com`.

[WKRQ+08] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, GeoffreyJ. McLachlan, Angus Ng, Bing Liu, PhilipS. Yu, Zhi-Hua Zhou, Michael Steinbach, DavidJ. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.

[WLA+09] Yi Wang, Jialiu Lin, Murali Annavaram, Quinn A. Jacobson, Jason Hong, Bhaskar Krishnamachari, and Norman Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 179–192, New York, NY, USA, 2009. ACM.

[WLTS06] J.A. Ward, P. Lukowicz, G. Troster, and T.E. Starner. Activity recognition of assembly tasks using body-worn microphones and accelerometers. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(10):1553–1567, Oct 2006.

[ZCL+10] Yu Zheng, Yukun Chen, Quannan Li, Xing Xie, and Wei-Ying Ma. Understanding transportation modes based on gps data for web applications. *ACM Trans. Web*, 4(1):1:1–1:36, January 2010.

[ZKS10] Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh. Improving energy efficiency of location sensing on smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 315–330, New York, NY, USA, 2010. ACM.